1988

AdaFlow: the automation of software analysis using Petri Nets.

Grecco, Albert J.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/22853

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ADAFLOW: THE AUTOMATION OF
SOFTWARE ANALYSIS USING PETRI NETS

by

Albert Joseph Grecco

June 1988

Thesis Advisor: Daniel L. Davis

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBERS(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (include Security Classification)
AdaFlow: The Automation of Software Analysis Using Petri Nets

12 PERSONAL AUTHOR(S)
Grecco, Albert J.

13a TYPE OF REPORT	13b. TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM ____ TO ____	1988, June	275

16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES

FIELD	GROUP	SUB - GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
AdaFlow; AdaMeasure; Ada; Petri Nets; software analysis; software metrics

19 ABSTRACT (Continue on reverse if necessary and identify by block number)
There is considerable interest in the the development of Ada-based, automated software tools to aid in the development and testing of embedded, real-time software. The Naval Postgraduate School has already implemented automated Ada metric tools at the request of the Naval Weapons Center, China Lake. This thesis is the preliminary work for a new automated software analysis tool entitled "AdaFlow". This tool, which is written in Ada, takes Ada programs as input, and translates the source code to a Petri net model. This Petri net model provides the user with the capability to perform automated, interactive analysis of Ada programs for properties such as safety and deadlocks. Recommendations for future work in this area are included.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED / UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. Daniel L. Davis	22b TELEPHONE (Include Area Code) (408) 646-3091 22c OFFICE SYMBOL Code 52Dv

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

* U.S. Government Printing Office 1986-606 24

UNCLASSIFIED

Approved for public release; distribution is unlimited

AdaFlow:
The Automation of
Software Analysis Using Petri Nets

by

Albert J. Grecco
Lieutenant, United States Navy
B. S. E. E., United States Naval Academy, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING SCIENCE

ABSTRACT

There is considerable interest in the the development of Ada®-based, automated software tools to aid in the development and testing of embedded, real-time software. The Naval Postgraduate School has already implemented automated Ada metric tools at the request of the Naval Weapons Center, China Lake. This thesis is the preliminary work for a new automated software analysis tool entitled "AdaFlow". This tool, which is written in Ada, takes Ada programs as input, and translates the source code to a Petri net model. This Petri net model provides the user with the capability to perform automated, interactive analysis of Ada programs for properties such as safety and deadlocks. Recommendations for future work in this area are included.

Ada is a registered trademark of the U. S. Government--Ada Joint Program Office

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to acknowledge the efforts of Karl Fairbanks, one of the original authors of 'AdaMeasure'. His testing of of the new Token Scanner in 'AdaMeasure' before 'AdaFlow' was operational and his knowledge of the Ada grammar kept me pointed in the right direction. A copy of the 'AdaMeasure' source code was provided by Karl and saved much wear and tear on the keyboard.

My wife, Lacy, has been a source of strength and inspiration throughout my studies. Although working and completing her own graduate studies, she was always able to take up the slack when I was tethered to the computer.

I. INTRODUCTION

A. ADA-BASED SOFTWARE TOOLS

As the Department of Defense's commitment to the Ada language is firm, there is considerable interest in the development of Ada-based, automated software tools. The purpose of these tools is to increase the productivity and efficiency of software engineering efforts. Ada-based, automated metric tools have been successfully implemented at the Naval Postgraduate School in response to this need and at the request of Naval Weapons Center, China Lake; specifically, Neider and Fairbank's implementation of the Halstead Length Metric in a thesis entitled "AdaMeasure" [Ref. 1], and Herzig's extension of "AdaMeasure" to include the Sallie Henry and Dennis Kafura Complexity Flow Metric [Ref. 2].

Rather than rely on a specific metric implementation, the design of "AdaMeasure" incorporates a general top-down, recursive descent parser to collect the desired metric information. This parser relies on the premise that the input code has been correctly compiled before being analyzed for the desired metric data. This assumption allows the parser to utilize a modified Ada grammar which reduces the size and complexity of the parser while retaining the capability to parse an input file in enough detail to collect meaningful and relevant metric data. [Ref 1:p. 28]

B. ANALYSIS OF REAL-TIME EMBEDDED SYSTEMS

Of the available methods for performing software analysis, Leveson and Stolzy [Ref. 3] advocate the use of Petri nets as the most viable method for conducting a systems approach to software analysis. They argue that a systems approach is required since real-time embedded software seldom works "in a vacuum". The choice of Petri nets as a desirable method for analysis is predicated on the ability of Petri nets to model hardware, software, and human behavior using the same language. An added advantage is that timing information can be incorporated into the Petri net model for analysis of real-time embedded systems. Leveson and Stolzy have proposed a Petri net based software analysis methodology that relies on deriving the untimed reachability graph of the system Petri net model in order to determine the timing constraints and properties of the final real-time imbedded system. Although principally concerned with software safety analysis, the analysis approach demonstrated by Leveson and Stolzy may be used to deduce other properties of a real-time embedded system. [Ref. 3]

Shatz and Cheng [Ref. 4] were the first to describe an automated, Petri net based method for static analysis of Ada programs. Their analysis approach consisted of the following three steps / subsystems as illustrated in Figure 1.1:

1. Translation of the source program into a Petri net model.
2. Analysis of the Petri net model.
3. Interpretation of the Petri net properties so as to derive properties of the source program. [Ref. 4:p. 378]

The Front End Translator Subsystem utilized a multi-pass translation algorithm and a translation table that stored Petri net equivalent templates of Ada structures of interest. As Shatz and Cheng were specifically concerned with distributed programs, their translation scheme concentrated on tasks and their synchronization and communication mechanisms. They did not explicitly consider Ada packages and function program units. These Petri net templates of Ada structures were uniquely labeled, linked together and related to source code on the second pass through the source code. This

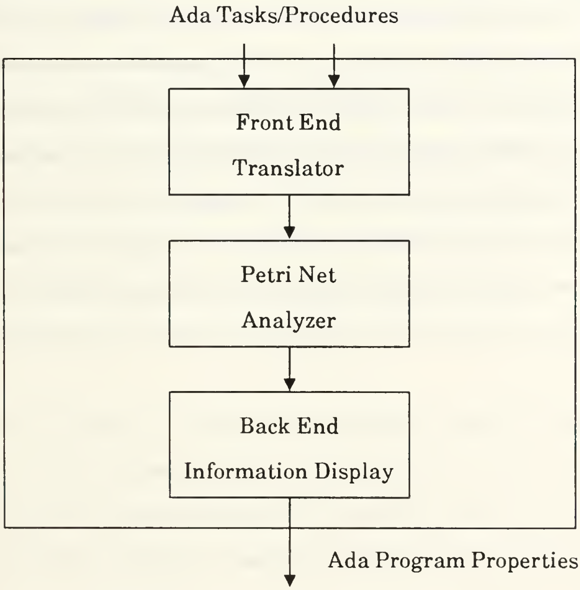


Figure 1.1 An Overview of the Shatz and Cheng Analysis System

"customization" of the templates was based on the premise that each statement had a unique statement number. [Ref. 4:pp. 378-380]

For the Petri Net Analysis Subsystem, Shatz and Cheng relied upon the P-NUT suite of tools provided by Rami R. Razouk of the University of California, Irvine. [Ref. 4:p.379]

The Back End Interpreter / Display Subsystem provided a metric report that related the results of the Petri net static analysis in the context of the source program so as to be an understandable and useful aid to the Ada programmer. [Ref. 4:p.378]

The software analysis methodology proposed by Leveson and Stolzy requires prior knowledge of the properties the programmer wants to analyze, e. g. , what constitutes a fault, failure, deadlock, etc. [Ref. 3:p. 1]. The incorporation of this preliminary analysis information into an automated software analysis tool suggests a capability to interactively query the Back End Interpreter / Display Subsystem rather than receive a canned metric product. These queries must be based upon knowledge, from either the programmer or the Interpreter Subsystem, of the source code to Petri net place mapping.

Although principally concerned with a distributed software system's potential communication patterns and complexity metrics [Ref 4.:p. 377; Ref. 5], Shatz and Cheng's concept of an automated petri net translator is ideally suited to the area of interactive software analysis. Unfortunately, the exclusion of key Ada constructs, the template implementation of the Front End Translator Subsystem, and the non-interactive Back End Interpreter /

Display Subsystem limits the usefulness of Shatz and Cheng's Analysis System as a practical interactive software analysis tool.

C. OBJECTIVES

It is the objective of this thesis to demonstrate and implement an algorithm for the automated translation of Ada source code to a Petri net model. This algorithm has an advantage over the template algorithm in that it requires only one pass through the source code. In addition, the intermediate products produced by this algorithm can facilitate the storing of libraries of source code Petri net models. This implementation of an automated Ada source code translator utilizes the same parsing technology of metrics developed at the request of Naval Weapons Center, China Lake and is intended to be the preliminary work for a new automated software analysis tool entitled "AdaFlow". Although "AdaFlow" is not intended to produce a metric product, it is designed to demonstrate the versatility of the "AdaMeasure" technology and to be the logical companion of the "AdaMeasure" metric product.

II. REVIEW OF THEORY

A. PETRI NETS

Petri nets were originally designed as a tool to model communication between asynchronous components of a computer system by Carl Petri [Ref. 6]. Petri nets have evolved as a modeling tool and have found application in such diverse areas of study as software, hardware, economics, and chemistry.

A formal definition of a Petri net is a five-tuple, $\Phi = (P, T, I, O, \mu_0)$, where:

1. $P = \{p_1, p_2, \cdots, p_n\}$ is a finite set of places and $n \geq 0$.
2. $T = \{t_1, t_2, \cdots, t_m\}$ is a finite set of transitions; $m \geq 0$; and the set of places and transitions are disjoint, $P \cap T = \varnothing$.
3. I is the input function $T \Rightarrow P^\infty$, a mapping from transitions to bags of places.
4. O is the output function $T \Rightarrow P^\infty$, a mapping from transitions to bags of places.
5. μ_0 is the initial marking for the net, $P \Rightarrow N$ where N is the set of nonnegative integers. [Ref. 3:pp. 396-397]

A graph structure is most often used to illustrate a Petri net. Standard symbols include a circle "O" to represent a place and a bar " | " to represent a transition. An arrow or arc from a place to a transition defines the place as an input to the transition while an arc from a transition to a place defines the place as an output to the transition as illustrated in Figure 2.1. [Ref 3:p. 387]

In order to illustrate the dynamic nature of a system being modeled, Petri nets utilize tokens. The initial marking, μ_0, deposits zero or more tokens in each Petri net place. This marking corresponds to the initial state of the system. The net is animated by the movement of tokens from input places,

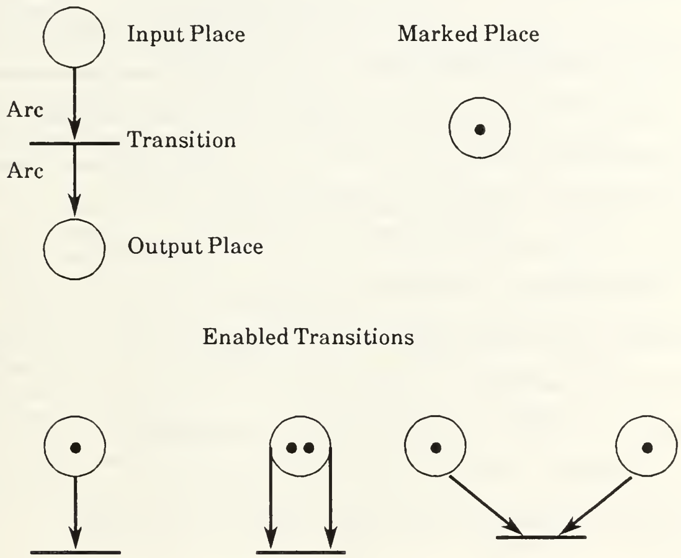


Figure 2.1 Standard Petri Net Symbology

through a transition, to output places. In order for a token to move, the transition separating source places and destination places must be enabled. A transition is enabled only if each input place to the transition contains at least as many tokens as there are arcs from the input place to the transition . Examples of enabled transitions are shown in Figure 2.1. In an untimed Petri

net, a transition may fire any time after it is enabled. When a transition fires, all tokens enabling that transition are removed from their corresponding input places and one token is deposited in each of the transition's output places. Transitions continue to fire as long as at least one transition remains enabled. [Ref. 3]

The initial state of the system is defined by the initial marking, μ_0. When a transition fires, the new marking of tokens defines a new system state. For an untimed Petri net, the enabled transitions may fire in any order. The set of all possible states that may exist based on all possible orderings of transition firings defines the reachable states for the system. In this thesis, a reachability graph will be used to illustrate the reachable states for a system.

A Time Petri net is a Petri net that is enhanced to include timing constraints on the firing of transitions. The addition of timing information may limit the reachable states of the system, but may never increase them. This principle is key to the analysis technique described by Leveson and Stolzy that begins with the untimed reachability states of a system and works backward to the real-time properties of a system. [Ref. 3:p. 389]

B. MODELING COMPUTER SOFTWARE

In his description of modeling with Petri Nets, Peterson claims that the modeling of computer software is "...perhaps the most common use of Petri nets and has the greatest potential for useful results." [Ref. 7:p. 54]

In modeling, a decision must be made concerning which aspects of the real system are to be incorperated into the model. When applied to computer

software, Petri net models best illustrate the aspect of software control structures. Peterson's rationale for modeling control structures is as follows:

> Petri nets are meant to model the sequencing of instructions and the flow of information and computation but not the actual information values themselves. A model of a system, by its nature, is an *abstraction* of the modeled system. As such it ignores the specific details as much as possible. If all the details were modeled, then the model would be a duplicate of the modeled system, not an abstraction. [Ref. 7:p. 55]

As flowcharts are a standard means of representing the control structures of a program, Peterson utilizes flowcharts as an intermediate form of the source code in the translation of concurrent computer software. In his description of the translation methodology, single processes in a system of concurrent processes are first described in terms of flowcharts. These flowcharts are translated to Petri nets, and then combined to yield one Petri net representation for a system of concurrent processes. [Ref. 7:pp. 54-68]

The translation of flowcharts to Petri nets relies on the similarities between these two graphical means of representating of a program. In a flowchart, nodes model actions or events, while arcs between nodes model conditions. In a Petri net, the transitions model actions, while nodes model conditions. Peterson's translation is, therefore, very straightfoward: replace the nodes of the flowchart with transitions in the Petri net and the arcs of the flowchart with places in the Petri net as illustrated in Figure 2.2. Peterson describes a one-to-one correspondence between flowchart arcs and Petri net places, while flowchart nodes are represented in different ways, depending on the type of the node: computation or decision [Ref. 7: p. 58]. The combining of Petri net models for single processes into one model representing a system of concurrent processes is accomplished by introducing the concept of parallelism and synchronization.

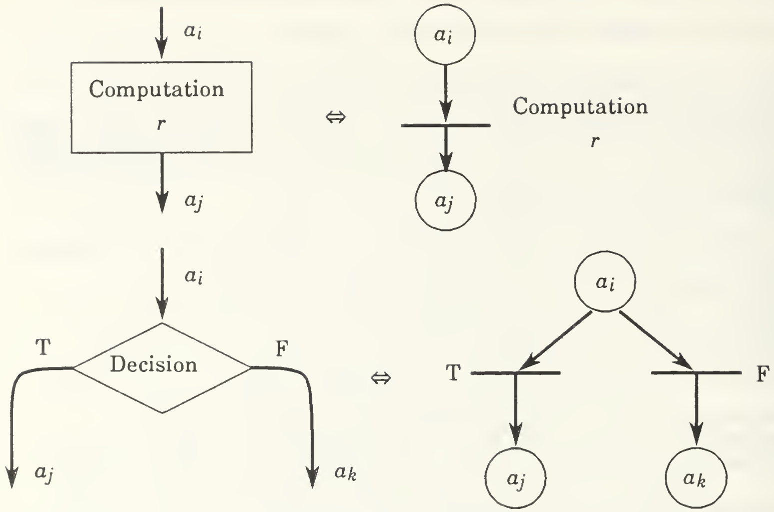


Figure 2.2 Translating Flowcharts to Petri Nets [Ref 7:p. 57]

Peterson describes three ways parallelism can be introduced into a software model:

1. Simply take the union of all Petri nets to represent the concurrent execution of each individual process. Each process has an initial marking in the place representing the initial program counter for that process.
2. Utilize the FORK and JOIN operations originally proposed by Dennis and Van Horn [Ref. 8]. The FORK and JOIN operations are illustrated in Figure 2.3.
3. Utilize the *parbegin* and *parend* control structures suggested by Dijkstra [Ref. 9]. This construct is illustrated in Figure 2.4.

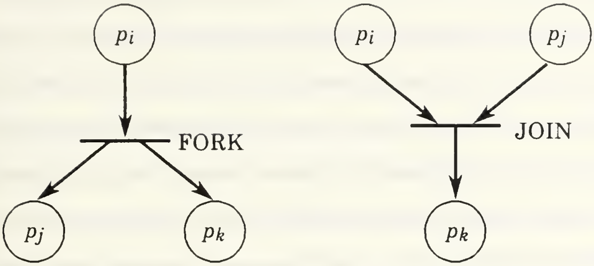


Figure 2.3 Modeling the FORK and JOIN Operations [Ref 7:p. 60]

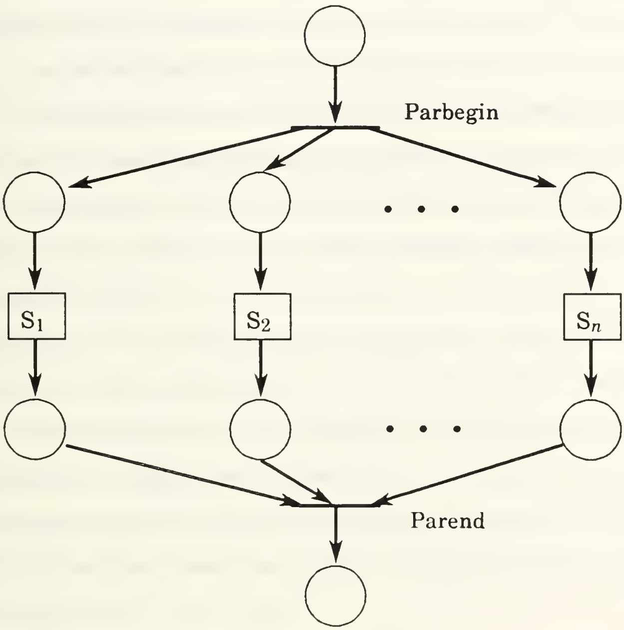


Figure 2.4 Modeling the *Parbegin* and *Parend* Operations [Ref 7:p. 61]

In his assessment of the first method, Peterson remarks that although it introduces a parallelism that cannot be represented in a flowchart, it is still not a very useful method of modeling parallelism [Ref. 7:p. 59]. The second method is a more accurate depiction of how parallelism would normally be introduced into a process in a computer system; however, it limits the number of processes that may be spawned to two. The *parbegin* and *parend* structure offers the accurate depiction of how parallelism would normally be introduced without the restriction on the number of processes that may be spawned [Ref. 7:pp. 59-61]

The concept of synchronization entails the sharing of information and resources between individual processes. This communication between processes must be restricted and coordinated in order to ensure correct operation of the overall system. Peterson describes classic synchronization problems such as the mutual exclusion problem [Ref. 10], the producer / consumer problem [Ref. 9], the dining philosophers problem [Ref. 9], and the readers / writers problem [Ref. 11], and presents some Petri net solutions to these problems. As these classic synchronization problems do not reflect the synchronization problems of a specific computer language, Peterson does not relate his solutions to a computer software translation algorithm. His solutions merely illustrate general methods for modeling general classes of synchronization problems. A discussion of Ada's synchronization mechanisms and a specific translation algorithm will be presented in Chapter III. [Ref. 7: pp. 61-69]

The procedure for modeling computer software outlined by Peterson relies on two translations: from source code to flowchart and from flowchart to Petri

net. In addition, one must then add Petri net details in order to model parallelism and synchronization mechanisms between the Petri nets produced from the two translations. Although this procedure will ultimately yield a Petri net model of the computer software under study, it is not a procedure that is readily automated. The modeling algorithm detailed by Shatz and Cheng, although specific to Ada software, overcomes this limitation by automating the translation process. This modeling algorithm required two steps:

1. Preprocessing of the source code which collects "necessary information" into some tables for later reference.
2. Translation of the source code. [Ref. 4]

The preprocessing step required one complete pass through the source code to build the tables required by the translator. As one example of what is considered "necessary information" for the preprocessor to collect, Shatz and Cheng describe the maintenance of an Entry Call Table. The Entry Call Table has four fields:

1. The name of the calling task.
2. The name of the called task.
3. The name of the entry in the called task.
4. A unique identifier for the entry call.

In order to uniquely identify entry calls and others information collected by the preprocessor, Shatz and Cheng assume each statement has a unique statement number. [Ref 4:p. 380]

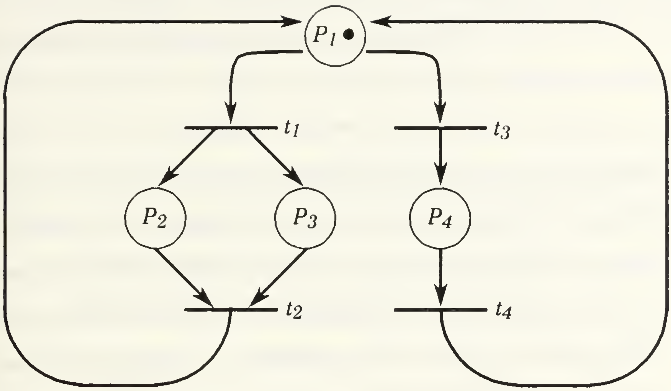
The translation phase of the algorithm required a second complete pass through the source code. The translator utilized a template table of stored

Petri net equivalent models of Ada control structures. These Petri net equivalent models and the resulting source program model were stored and described in terms of a Petri net abstract grammar. As defined by Shatz and Cheng, a Petri net abstract grammar is a triple AG = (P, T, PR), where:

1. P is a finite set of non-terminal symbols that correspond to places in the Petri net.
2. T is a finite set of terminal symbols that correspond to transitions in the Petri net.
3. PR is a finite set of production rules of the form $u \Rightarrow tv$, where u and v are strings of symbols from P, and t is a symbol from T.

An initial string is used to represent the initial marking of the Petri Net. Figure 2.5 illustrates an example Petri net model and the corresponding abstract grammar representation. [Ref. 4:pp.378-379]

The process of translating Ada constructs consisted of retrieving the appropriate Ada construct model from the template table, customizing the templates, and linking the templates together. Customizing the templates not only uniquely identifies places within the templates, it also provides the means to automate the modeling of synchronization mechanisms between Petri net models of single processes. Consider the example of Figure 2.6. Shatz and Cheng's templates for Ada's *entry* statement and *accept* statement are shown before customization. Customization results in the Ack-entry place for both templates receiving the same unique identifier. Therefore, in the abstract grammar representation, these two building blocks of Ada's synchronization mechanism are effectively linked. [Ref. 4:p. 380]



1. P1 ⇒ t1 P2 P3
2. P2 P3 ⇒ t2 P1
3. P1 ⇒ t3 P4
4. P4 ⇒ t4 P1

with initial string = P1

Figure 2.5 An Abstract Grammar Representation of a Petri Net Model [Ref. 4:p. 384]

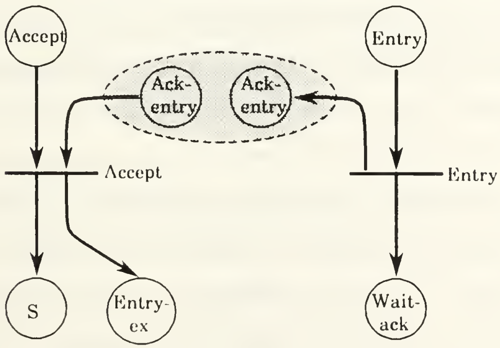


Figure 2.6 Modeling Ada's Synchronization Mechanism

This algorithm for modeling computer software is superior to Peterson's algorithm. Although automated, there exist some notable shortcomings that prevent the use of this template algorithm in a general, automated, Ada software analysis tool. These shortcomings include:

1. The algorithm requires multiple passes through the source code. The first pass is utilized to determine the underlying structure of the program, while the second pass effects the actual translation.
2. The tables assembled in the first pass do not include scoping information and ,therefore, do not present a true picture of the program's underlying structure. In a general Ada program, *with* and *use* clauses can dramatically alter the context of compilation and provide direct visability to identifiers without using the "dot" or component select notation. If the tables are unable to provide scoping information, the constuct being modeled may be misidentified.
3. The method used to depict parallelism is to provide an initial marking for the main procedure and each task in the source code. This is not an accurate description of of how parellelism would normally be introduced into a process. A more accurate depiction would utilize the *parbegin* and *parend* structures.
4. The assumption of unique statement numbers is, perhaps, unrealistic. If by "statement number", one refers to the line of text in the source code where the statement is physically located, then the translation algorithm imposes restrictions on the language beyond those of the Language Reference Manual (LRM) [Ref. 12].

5. The use of templates is a rigid method that does not accurately depict the flow of control in a general Ada program.

C. FRONT-END MACHINE

Rather than rely on a tool that was only capable of gathering specific metric information, Neider and Fairbanks chose to develop a generic Ada front-end machine consisting of a lexical analyzer and parser. This front-end machine was used to construct an intermediate representation of the source program, or derivation tree, which is utilized to collect the information necessary to implement the desired metric. [Ref. 1:p. 18]

As this derivation tree determined the underlying structure of the program incrementally, while the program was being scanned, the desired metric information could be collected in one pass through the source code. This is accomplished by effecting emissions of the desired information from the front-end machine at appropriate places in the derivation tree. By altering these emissions from metric information to Petri net information, the front-end machine can be utilized to translate Ada source code to Petri net models.

1. The Modified Ada Grammar

Nieder and Fairbanks decided on a top-down, recursive-descent parsing algorithm as the implementation of the parser. Recursive-descent parsers are closely related to the LL(1) subset of the context-free grammars and are among the most popular of the compiler parsers [Ref. 13:p. 167]. For this reason, it was necessary to "massage" the Backus-Naur description of the Ada language [Ref. 12:Appendix E], a non-LL(1) grammar, into an LL(1)-like

grammar capable of being parsed deterministically. In the context of this thesis, "massage" refers to the process of removing instances of left recursion and then left factoring the grammar so the parser can choose the correct production rule based on one token look-ahead. [Ref. 1:p. 13]

Nieder and Fairbanks discovered several instances of left-recursion in the Ada grammar. The following excerpt from their thesis illustrates Ada's left-recursive quality for the production rule NAME. Ada's terminal tokens will appear in lower case letters while nonterminals will appear in upper case letters:

> The production rules, when taken directly from the LRM, appear as follows:
>
> NAME ⇒ identifier
> ⇒ character__literal
> ⇒ string__literal
> ⇒ INDEX__COMPONENT
> ⇒ SLICE
> ⇒ SELECTED__COMPONENT
> ⇒ ATTRIBUTE
>
> INDEXED__COMPONENT ⇒ PREFIX (EXPRESSION)
>
> SLICE ⇒ PREFIX (DISCRETE__RANGE)
>
> SELECTED__COMPONENT ⇒ PREFIX . SELECTOR
>
> ATTRIBUTE ⇒ PREFIX ' ATTRIBUTE__DESIGNATOR
>
> PREFIX ⇒ NAME
> ⇒ FUNCTION__CALL
>
> When starting with NAME and substituting in the productions, the left recursion becomes readily apparent. For example:
>
> NAME ⇒ SLICE ⇒ PREFIX(DISCRETE__RANGE) ⇒ NAME(DISCRETE__RANGE). [Ref. 1:pp. 14-15]

These instances of left recursion required extensive massaging in order to yield an LL(1) grammar. The resulting grammar is included as Appendix A.

2. Lexical Analysis

The task of assembling a sequence of source characters into the terminal alphabet or *tokens* of the language is within the province of the scanner or lexical analyzer [Ref. 13:p. 18]. There are seven classes of tokens that comprise the terminals of the Ada language. These token classes are known as *identifiers*, *separators*, *numeric literals*, *delimiters*, *comments*, *character literals*, and *string literals*. In addition, the Ada language recognizes a special sub-class of *identifier* known as *reserved words*.

The process of lexical analysis entails reading the source program one character at a time and building the tokens deterministically, with one character look-ahead, based upon the definition of Ada's lexical elements as described in Chapter Two of the LRM [Ref. 12].
Neider and Fairbanks described seven deterministic finite state machines capable of recognizing the seven basic token classes of the Ada language. These machines will be discussed in greater detail in Chapter III. [Ref. 1:pp. 18-25].

3. Recursive-Descent Parser

The implementation of Neider and Fairbanks' recursive-descent parser consists of a set of function calls with a one-to-one correspondence to the non-terminals of the Modified Ada Grammar. These function calls return either a true or false value. A return of false excludes the non-terminal from the derivation tree while a return of true indicates that the non-terminal is part of the derivation tree. As non-terminals may contain *tokens* as part of the production string, the parser can query the lexical analyzer if the current token matches a terminal in the production string. If a match is found, the

token becomes a leaf of the derivation tree and a new token is assembled by the lexical analyzer. Parsing begins with a call to the function COMPILATION, the starting non-terminal of the grammar [Ref. 1].

III. THE METAMORPHOSIS OF "ADAMEASURE"

"AdaMeasure" is an evolving metric tool that is utilized and maintained by the Software Missile Branch of the Naval Weapons Center, China Lake. Since it was first published in March of 1987, The "AdaMeasure" front-end machine has undergone a significant change in appearance while retaining it's basic functionality. During the course of this thesis, several changes to the lexical analyzer and the Modified Ada Grammar were proposed and incorporated. Changes to the lexical analyzer were made primarily in the interest of speed and readability, while changes to the Modified Ada Grammar were made primarily in the interest of regularity. The first two sections of this chapter outline these general modifications, while the last section details the changes made in the Parser (Appendix C) emissions in order to realize a Petri net model of the source code.

A. LEXICAL ANALYZER

Prior to this thesis, many of the functional tasks of lexical analysis were interspersed throughout the different packages that comprised the front-end machine. This thesis sought to group all the functional tasks of lexical analysis into one package with an interface that hides the implementation details as much as possible. The result of this effort is the Token Scanner package.(Appendix H). This package presents an interface that, to the user, makes the source file appear as a logical file of Ada tokens. A finite set of operations are provided to the user that include the ability to view the token

under the read head, view the token that will come under the read head next, and the ability to advance the read head one token at a time. In addition, the capabilities of the Token Scanner were expanded to include the capability to distinguish *reserved words* from *identifiers*. This change allowed an efficient hash search for reserved words that was hidden from the user, and resulted in a significant increase in speed for the front-end machine.

The implementation of the Token Scanner utilizes a pipe to assemble the tokens of the language and a filter to prevent *comments* and *separators* from ever coming under the read head or into the look-ahead position. The seven deterministic finite machines described by Nieder and Fairbanks [Ref. 1] are utilized in the pipe to identify the tokens as they are assembled. These machines have been enhanced to conform more closely to the exact lexical requirements of the LRM. The only lexical requirement the Token Scanner does not enforce, is the requirement that each extended digit of a based *numeric literal* be less than the base [Ref. 12:p. 2-5]. These enhancements have virtually eliminated the Token Scanner's reliance on the precondition that the source code be correctly compiled prior to being analyzed.

B. GRAMMAR

As this thesis progressed, it became apparent that there were many productions in the Modified Ada Grammar that could be simplified. Consider the original productions that were designed to parse an Ada function:

```
FUNCTION_UNIT ⇒ DESIGNATOR FUNCTION_UNIT_TAIL

FUNCTION_UNIT_TAIL ⇒ is new NAME [GENERIC_ACTUAL_PART ?] ;
                   ⇒ [ FORMAL_PART ?] return NAME FUNCTION_BODY
```

FUNCTION_BODY ⇒ is [FUNCTION_BODY_TAIL ?]
⇒ ;

FUNCTION_BODY_TAIL ⇒ separate ;
⇒ < > ;
⇒ SUBPROGRAM_BODY
⇒ NAME ;

These productions were simplified to the following production rule:

FUNCTION_UNIT ⇒ DESIGNATOR [FORMAL_PART ?] return NAME is SUBPROGRAM_BODY
⇒ DESIGNATOR [FORMAL_PART ?] return NAME ;
⇒ DESIGNATOR [FORMAL_PART ?] return NAME renames NAME ;
⇒ DESIGNATOR is SUBPROGRAM_BODY

Another significant change in the grammar concerned the production rules for SUBPROGRAM_BODY. There were numerous instances of productions requiring the sequence:

[DECLARATIVE_PART ?] begin SEQUENCE_OF_STATEMENTS [exception [EXCEPTION_HANDLER]$^+$?] end [DESIGNATOR ?] ;

Rather than duplicate this sequence for each production, the productions requiring this sequence were modified to utilize the SUBPROGRAM_BODY production rules. This simplification relies on the precondition of correctly written code verified by a compiler prior to being analyzed. The Modified Ada Grammar listed in Appendix A contains all the changes to the original grammar and is the current grammar utilized in both "AdaMeasure" and "AdaFlow".

C. PARSER EMISSIONS

1. Code Blocks

A key issue in any source code to Petri net translation algorithm is the method used for transforming source code space into Petri net space. Shatz and Cheng [Ref. 4] chose to use "statement numbers" that corresponded

to the line of text in the source code where the statement was physically located. This method of transformation assumes that each Ada control structure has a unique statement or line number. This assumption is unrealistic as it imposes restrictions on the language beyond those of the LRM.

One method of transforming source code space to Petri net space is suggested by the very aspect of computer software Petri nets model best: control structures. Software control structures not only correspond to transitions in a Petri net, they also serve to separate source code into "blocks" of code that correspond to unique Petri net places. It is not sufficient, however, to rely on control structures as the only demarcation of where these code blocks begin and end. One must also consider the possible source code destinations that a control structure can transition to when executed. These possible destinations include *labels*, *procedures*, *functions*, and *task entries*. In general, a control structure is located in the current code block and denotes the end of that code block, while a destination denotes the end of the current code block and is located in the next code block. The execution of control structures is simply the order in which these code blocks are interconnected.

Consider the simple Ada program and corresponding Petri net places of Figure 3.1. The procedure entitled MAIN defines a destination of a procedure call statement and, therefore, begins a new code block. A procedure is a scope defining construct that, when viewed from the perspective of the invoker, can be considered as one large code block or a *super-place* in the corresponding Petri net. The details of control flow internal to the procedure are not visible to the outside world. All the declarations that follow MAIN are

```
procedure MAIN is
type GRADE__BOOK is array (positive range 1..10) of
natural;
INDEX : natural;
TOTAL : natural ;
AVERAGE : natural;
STUDENT : GRADE__BOOK;
begin
     INDEX := 0;
     TOTAL := 0;
     <<ADD__AGAIN>>
          INDEX := INDEX + 1;
          TOTAL := TOTAL +
          STUDENT(INDEX);
          if (INDEX = 10) then
               goto CONTINUE;
          else
               goto ADD__AGAIN;
          end if;
     <<CONTINUE>>
          AVERAGE := TOTAL/10;
end MAIN;
```

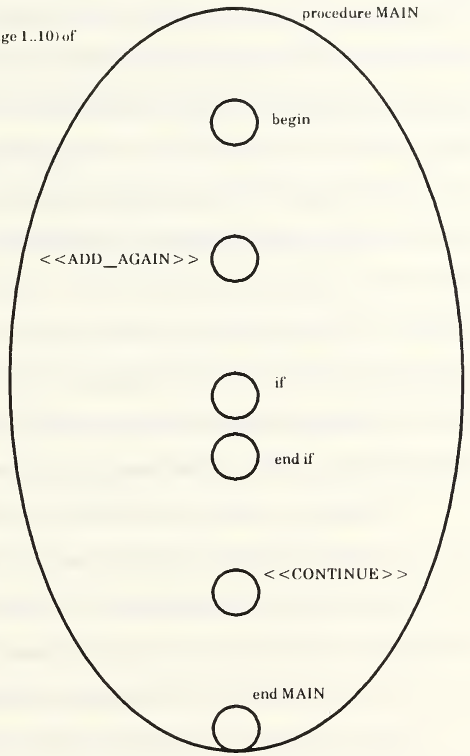


Figure 3.1 Transforming Source Code Blocks to Petri Net Places

within the same code block as MAIN. The reserved word *begin* labels the start of MAIN's internal control structure and starts a new code block. The label ADD__AGAIN ends the first internal code block and is located in the next code block. The *if* statement labels the root location of a multi-way decision path and, therefore, is the beginning of a new code block. The first path of the if statement is an unconditional jump to the label CONTINUE. This

statement is part of, and denotes the end of, the *if* code block. The *else* clause of the *if* statement reactivates the root location as the current code block. The *goto* statement of the second path has the same effect on the *if* code block as the *goto* of the first path. The *end if* statement is a possible destination for any of the paths of the *if* statement and, as such, denotes the end of the code block in the current path if it has not already ended. The *end if* statement begins, and is located in, a new code block. The CONTINUE label ends the *end if* code block and is located in the next code block. The end of procedure MAIN labels a possible destination for control statements such as *return*; therefore, it denotes the end of the current code block and is the first statement in the next code block. Upon completing the parse of MAIN's subprogram body we exit the last internal code block and the enclosing procedure code block.

A necessary condition for translation is that for every code block in the source program, there must exist a unique Petri net place. This property is not commutative as *pseudo*-places exist in Petri nets that have no corresponding code blocks in the source program. These *pseudo*-places will be discussed when we consider the Parser's emissions for Petri nets.

Due to the front-end machine's ability to determine the deep, underlying structure of Ada programs, it is possible to determine when a code block, and the related Petri net place, begins and ends on the basis of where we are in the grammar rather than where we are in a text file. Based on this determination, the Parser emits information to the Code Blocker (Appendix F).

The Code Blocker is responsible for assigning a unique Petri net place number to each code block that is entered by the Parser. In addition, the code

blocker accepts and stores information from the Parser that relates the Petri net places to their locations in the text file. Although not currently used by the system, this information is maintained for two reasons:

1. It is easier for the user to relate Petri net places to source code locations rather than grammar locations.
2. It is anticipated that, at a later date, an interactive, high level user interface will be incorporated that will require this mapping information.

2. Symbol Table

Simply stated, the function of a symbol table is to store and retrieve identifiers and their associated properties. There are two properties of interest for a source code to Petri net translator: an identifier's attribute and location.

An identifier's attribute or classification is used to determine whether the identifier is a control structure or a possible destination of executing a control structure. If a control structure, the attribute uniquely classifies the type of control structure that will later be modeled. The attribute also determines whether or not the identifier is the beginning of a new scope.

As Ada is a statically scoped language with strict visibility rules, any symbol table used with Ada must preserve this scoping information. In addition, an Ada symbol table must allow for the capability to provide visibility of identifiers in previously exited scopes. This requirement is a by-product of Ada's package facility.

Symbol table location information, as it applies to a Petri net translator, relates the identifier to a unique code block and, therefore, a unique Petri net place. As an identifier may be declared before the location or code block is known, the capability to update an identifier's location must be supported by the symbol table.

By utilizing the location information from the Code Blocker, the front-end machine has all the additional resources required to manage the Symbol Table (Appendix E). Returning to the example of Figure 3.1, and ignoring the Parser's management of the Code Blocker for entering, exiting, and reactivating code blocks, the Parser's management of the Symbol Table can be illustrated.

When the Parser encounters the identifier MAIN, it obtains the current code block number from the Code Blocker, say "1", and inserts the identifier into the Symbol Table with a procedure declaration attribute and a location of "1". As a procedure declaration is a scope defining construct, this action causes the Symbol Table to enter a new scope.

The sequence of statements within a procedure body may contain a return statement. A return statement is used to complete the execution of the innermost enclosing procedure and may be thought of as an unconditional transfer to the end of the procedure. For this reason, the Parser makes an entry in the symbol table for the last code block in the procedure with a label attribute and a location of "0" or undefined. As each label in Ada must have a unique identifier, the reserved word *end* is used as the identifier for the last code block in MAIN. This method of labeling destination code blocks that do not have a user defined label ensures uniqueness and avoids clashes with user

defined labels as programmers are restricted from using a reserved word as a label identifier.

The next identifier that results in a Symbol Table entry is the label ADD_AGAIN. The Parser inserts ADD_AGAIN with a label attribute and the code block location, now "3".

Upon parsing the *if* statement, the Parser inserts the identifier *if* in the Symbol Table with a special attribute that identifies the *if* control structure and the location "4". This attribute causes the Symbol Table to enter a new scope. The Parser then inserts the *if* statement's corresponding, undefined *end* label.

The *goto* statement of the first *if* statement path causes the Parser to search the Symbol Table for the identifier CONTINUE. When the Symbol Table informs the Parser that CONTINUE is not declared, the Parser assumes that the *goto* statement is an implicit declaration of the label CONTINUE. This causes the Parser to insert a label for CONTINUE with an undefined code block location in the Symbol Table. The *goto* statement of the second *if* statement path causes the Parser to search the Symbol Table for the identifier ADD_AGAIN. The Symbol Table finds the label and reports this fact to the Parser. The Parser then checks to see if the location is defined (non-zero). If not defined, the Parser would update the Symbol Table entry to the current code block number.

The *end if* statement results in the Parser ordering the Symbol Table to search for the *end* label. When the Symbol Table finds the *end* label, the Parser then updates the label's location to the correct code block number of "5" and orders the Symbol Table to exit the scope.

When the CONTINUE label is encountered, the Parser orders the Symbol Table to search for the identifier CONTINUE. The Symbol Table finds the label and reports this fact to the Parser. The Parser then updates the label's location to the current code block number of "6".

The *end MAIN* statement results in the Parser ordering the Symbol Table to search for the *end* label. When the Symbol Table finds the *end* label, the Parser then updates the label's location to the correct code block number of "7" and orders the Symbol Table to exit the scope. Figure 3.2 illustrates the scoped symbol table at the end of the parse.

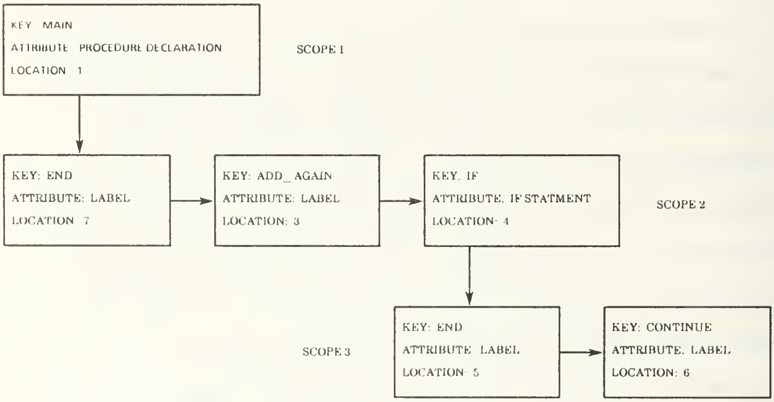


Figure 3.2 Storing Source Code Blocks in a Symbol Table

Ada supports the capability for a programmer to declare and invoke procedures, function, packages, tasks and entries before their corresponding bodies have been parsed. This capability is akin to the Pascal forward declaration. In order to handle these forward declarations, the Parser inserts the identifier, the appropriate declaration attribute, and an unknown

location. The Parser then inserts the corresponding *end* label with an unknown location and exits the scope. When the declaration's corresponding body is parsed, the Parser inserts the same identifier, with the appropriate body attribute, and the known code block location. This causes the Symbol Table to automatically search for and update the environment of definition, and enter that environment's scope.

3. Petri Net Transitions

Petri net transitions model the execution of control structures and connect Petri net places. Petri net places can be the source or destination of a transition For the purpose of this thesis, Petri net places will be divided into three categories: known Petri net places, unknown Petri net places, and *pseudo*-places. Known Petri net places correspond to the code block that is currently being parsed, while unknown Petri net places correspond to either a code block declared in the symbol table, or the next code block to be encountered. In all cases, known and unknown Petri net places correspond to a unique code block in the source. *Pseudo*-places are Petri net places that are required to model a control structure but have no corresponding location in source code. As an example of all three places, consider Figure 3.3 and the depiction of Ada's synchronization mechanism. When an entry to a task is called, the procedure that called the entry waits at the rendezvous until the invoked task accepts the entry and finishes processing the accept statements. Only then can the procedure that called the entry continue processing. Figure 3.3 depicts the two transitions required to model this control structure. The current code block is known by the Parser when the entry call statement is encountered. If the assumption that this is a correct Ada program is true,

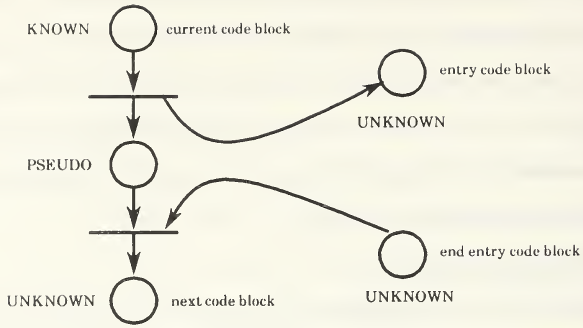


Figure 3.3 Known Places, Unknown Places, and Pseudo-Places

then the task specification must have been parsed and at least the entry code block and the corresponding end entry code block are in the Symbol Table. It is not necessary for the locations to be known yet. In order to model the requirement for the invoking procedure to wait at the rendezvous until the accept statements of the entry are through being processed, it is necessary to use a *pseudo*-place that has no corresponding code block in source code. The second transition models the completion of the entry. The token from the *pseudo*-place and the end entry code block act together to enable the transition for the invoking procedure to continue processing.

In this translater, the Parser emits known and unknown Petri net place information together with the type of control structure to be modeled to the Net Generator (Appendix D). For known Petri net places, the Parser emits the current code block number as provided by the Code Blocker. For unknown Petri net places, the Parser emits a pointer or access to the appropriate code block's entry in the Symbol Table. The Net Generator is

responsible for translating the control structure information into transitions between the known and unknown Petri net places. In addition, when it is necessary to use a *pseudo*-place to realize a model, the Net Generator grabs a unique location from the Code Blocker. During the course of this thesis, *psuedo*-places were only found necessary to realize models for procedure calls and entry calls. All other control structures were capable of being modeled by transitions between known and unknown Petri net places.

One special control structure is used so often it deserves special mention. In the Net Generator, this special control structure is called CONNECT_BLOCKS. Consider Figure 3.4 which represents the complete Petri net model for the previous example of Figure 3.1. The label ADD_AGAIN, although it signifies a possible destination of a control structure's execution, does not constitute a break in the sequential execution of MAIN. As the Parser knows the location associated with the *begin* code block, and the location associated with the ADD_AGAIN code block. The Parser simply emits these two known Petri net places to the Net Generator with the special control structure CONNECT_BLOCKS.

The Net Generator stores the Petri net model in an abstract representation similiar to the abstract grammar described by Shatz and Cheng [Ref. 4] . The reason for utilizing an intermediate representation of the Petri net model is to give the Symbol Table and Parser an opportunity to resolve unknown Petri net places. By storing access variables to the unknown Petri net places in the Symbol Table as part of the abstract representation of the Petri net model, the Symbol Table will automatically update the location of unknown Petri net places referenced in the Net Generator. For the

```
procedure MAIN is
type GRADE_BOOK is array (positive range 1..10) of
natural;
INDEX : natural;
TOTAL : natural ;
AVERAGE : natural;
STUDENT : GRADE_BOOK;
begin
      INDEX := 0;
      TOTAL := 0;
      <<ADD_AGAIN>>
            INDEX := INDEX + 1;
            TOTAL := TOTAL +
            STUDENT(INDEX);
            if(INDEX = 10) then
                  goto CONTINUE;
            else
                  goto ADD_AGAIN;
            end if;
      <<CONTINUE>>
            AVERAGE := TOTAL / 10;
end MAIN;
```

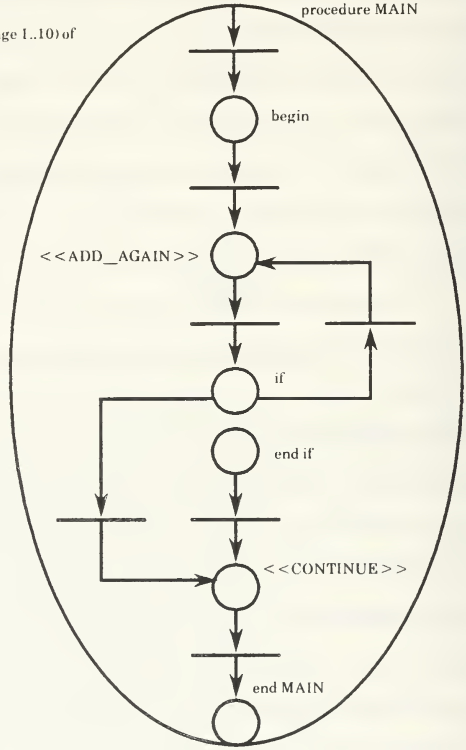


Figure 3.4   Transforming Control Structures to Transitions

unknown places that signify the next code block to be encountered, the Net Generator simply waits for the Parser to emit the next control structure. If the preceding model has an abstract representation that ends with an unknown place that is not a Symbol Table code block, the Net Generator chooses the next known code block location from the next Parser emission. As a correct Ada program is assumed and the question of Ada's separate

compilation facility has not as yet been addressed , all unknown Petri net places must be resolved by the end of the source code's parse. Only when the unknown places are resolved can we hope to generate a valid Petri net model of the source code.

Another reason for utilizing an intermediate representation of the Petri net model is that different Petri net analyzers may require a different specific input language. By simply adding a translation algorithm to the Net Generator, the abstract representation of the model can be translated to various Petri net analyzer input languages. The Net Generator has one translator already defined for the P-NUT set of tools [Ref. 14].

IV. "ADAFLOW"

"AdaFlow" is a concept for a Petri net based, interactive Ada program analyzer. This preliminary work concentrates on, and suggests a methodology for, the automatic production of Petri net models of Ada programs. The products of this translation method have been tailored to conform to the input format of an existing Petri net analyzer entitled P-NUT. The first section of this chapter briefly describes the P-NUT suite of tools and the capabilities these tools offer. The following sections of this chapter describe in detail the products produced by the translator and the environment in which the translator and P-NUT perform.

A. THE ANALYZER

P-NUT is a set of tools developed by the Distributed Systems Project in the Information and Computer Science Department of the University of California, Irvine. The tools were constructed primarily to assist researchers in applying Petri net analysis techniques in the design of distributed systems. The P-NUT suite of tools creates and manipulates three types of objects: Petri nets, reachability graphs and execution traces.

Petri nets are input to the system in textual form and are transformed by P-NUT into an internal representation of a Petri net. It is the function of the translator to provide the Petri net in this textual form. For a complete discussion of P-NUT's input language, the reader is referred to Reference 14.

Reachability graphs represent the state-space of a Petri net while execution traces represent portions of the state space. P-NUT has the capability to produce, analyze and display both timed and untimed reachability graphs from the internal representation of a Petri net. P-NUT also allows an execution trace to be converted into a partial reachability graph which can be analyzed and displayed in the same manner as a reachability graph produced from the internal representation of a Petri net.

The most powerful and innovative tool in P-NUT is a tool entitled Reachability Graph Analyzer (RGA) (Ref. 15). RGA reads the internal representation of a Petri net and its associated reachability graph and allows the user to do computer-assisted, interactive analysis, or "ask questions" about the model, using the language of first order predicate calculus with the addition of branching-time temporal logic operators. This interactive analysis capability is ideally suited to the concept of "AdaFlow".

B. THE TRANSLATOR PRODUCT

The following example demonstrates the modeling capabilities of the proposed translation method by producing a simple railroad crossing model similar to the model analyzed by Leveson and Stolzy [Ref. 3].

Figure 4.1 illustrates the original model used by Leveson and Stolzy to demonstrate their technique for analysis of real-time systems. Although there is no combination of Ada control structures that can exactly duplicate the places and transitions of the model in Figure 4.1 the following Ada program realistically portrays how an Ada task may be written to handle such a problem:

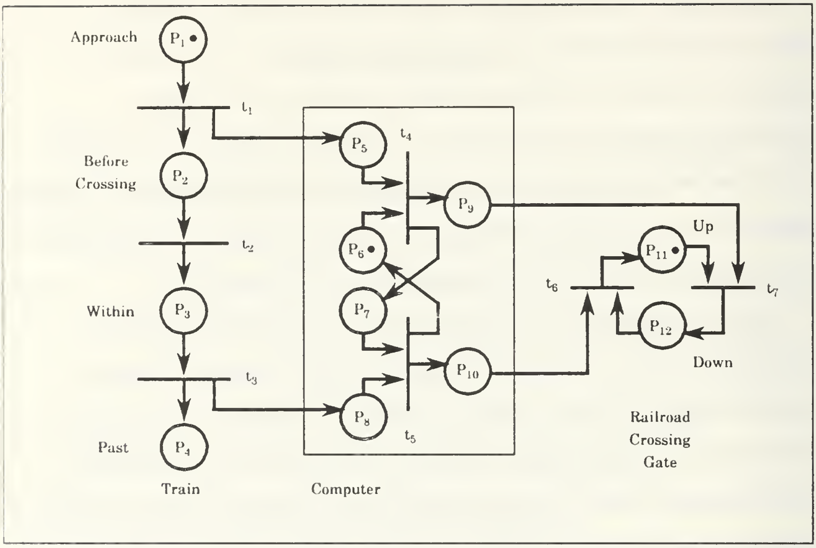


Figure 4.1 A Petri Net Model of a Simple Railroad Crossing

```
procedure RAIL__ROAD__CROSSING is
   task COMPUTER is
      entry APPROACH;
      entry DEPART;
   end COMPUTER;
   task GATE__KEEPER is
      entry LOWER__GATE;
      entry RAISE__GATE;
   end GATE__KEEPER;
   task body COMPUTER is
   begin
      loop
         accept APPROACH do
            null;
         end APPROACH;
         GATE__KEEPER.LOWER__GATE;
         accept DEPART do
            null;
         end DEPART;
         GATE__KEEPER.RAISE__GATE;
      end loop;
   end COMPUTER;
task body GATE__KEEPER is
   begin
      loop
         accept LOWER__GATE do
            null;
         end LOWER__GATE;
         accept RAISE__GATE do
            null;
         end RAISE__GATE;
      end loop;
   end GATE__KEEPER;
begin
   COMPUTER.APPROACH;
   <<BEFORE__CROSSING>> null;
   <<WITHIN__CROSSING>>
   COMPUTER.DEPART;
   <<PAST__CROSSING>> null;
end RAIL__ROAD__CROSSING;
```

The task entitled COMPUTER represents the software for the railroad crossing system, while the task entitled GATE_KEEPER and the main procedure represent a test harness for the COMPUTER software.

Assuming that this program is stored in a file entitled TRAIN2.ADA, a typical session with the "AdaFlow" translator would begin:

```
WELCOME TO ADAFLOW

ENTER THE NAME OF AN ADA SOURCE FILE TO MODEL
```

The user would respond with TRAIN2.ADA. The "AdaFlow" translator would notify the user:

```
PARSING BEGINS . . .
```

When "AdaFlow" has finished the translation, it gives the final message:

```
. . . PARSE SUCCESSFUL
```

and exits to the operating system. "AdaFlow" creates two files. The first file is named A.OUT and it contains the Petri net model of the source code written in the P-NUT input language. The second file, PLACE.DAT, is provided for the user to relate Petri net places to lines of text in the source code. For the Ada program stored in TRAIN2.ADA, the A.OUT file would appear as:

```
:t1: p1 -> p2, p3, p19
:t2: p3 -> p4
:t3: p4 -> p5
:t4: p6, p5 -> p7
:t5: p7 -> p8, p9
:t6: p9 -> p22, p10
:t7: p24, p10 -> p11
:t8: p12, p11 -> p13
:t9: p13 -> p14, p15
:t10: p15 -> p26, p16
:t11: p28, p16 -> p17
:t12: p17 -> p5, p18
:t13: p19 -> p20
:t14: p20 -> p21
:t15: p22, p21 -> p23
:t16: p23 -> p24, p25
:t17: p26, p25 -> p27
:t18: p27 -> p28, p29
:t19: p29 -> p21, p30
:t20: p2 -> p31
:t21: p31 -> p6, p32
:t22: p8, p32 -> p33
:t23: p33 -> p34
:t24: p34 -> p12, p35
:t25: p14, p35 -> p36
:t26: p36 -> p37
:t27: p30, p18, p37 -> p38
<p1>
```

The PLACE.DAT file relating locations in the source code to Petri net places would appear as:

LOCATION	CODE BLOCK LABEL	STARTING LINE	ENDING LINE
p1	START	0	0
p2	PROCEDURE CODE BLOCK	1	40
p3	TASK CODE BLOCK	10	22
p4	BEGIN SUBPROGRAM	11	12
p5	LOOP BLOCK	12	13
p6	ENTRY BLOCK	13	13
p7	BEGIN ACCEPT STATEMENTS	13	14
p8	END ENTRY BLOCK	15	15
p9	ENTRY CALL	15	16
p10	WAIT RENDEZVOUS	0	0
p11	ACCEPT STATEMENT	17	17
p12	ENTRY BLOCK	17	17
p13	BEGIN ACCEPT STATEMENTS	17	18
p14	END ENTRY BLOCK	19	19
p15	ENTRY CALL	19	20
p16	WAIT RENDEZVOUS	0	0
p17	END LOOP	21	21
p18	END SUBPROGRAM	22	22
p19	TASK CODE BLOCK	23	33
p20	BEGIN SUBPROGRAM	24	25
p21	LOOP BLOCK	25	26
p22	ENTRY BLOCK	26	26
p23	BEGIN ACCEPT STATEMENTS	26	27
p24	END ENTRY BLOCK	28	28
p25	ACCEPT STATEMENT	29	29
p26	ENTRY BLOCK	29	29
p27	BEGIN ACCEPT STATEMENTS	29	30
p28	END ENTRY BLOCK	31	31
p29	END LOOP	32	32
p30	END SUBPROGRAM	33	33
p31	BEGIN SUBPROGRAM	34	35
p32	WAIT RENDEZVOUS	0	0
p33	LABELLED BLOCK	36	37
p34	LABELLED BLOCK	37	38
p35	WAIT RENDEZVOUS	0	0
p36	LABELLED BLOCK	39	39
p37	END SUBPROGRAM	40	40
p38	STOP	0	0

The places that have a STARTING LINE and ENDING LINE of "0" are pseudo-places manufactured by the Net Generator.

Figure 4.2 illustrates the Petri net model of the train crossing produced by AdaFlow. By including a software test harness, a Petri net model for the software and the software's environment was realized. This model is significant in that it is capable of system's level, automated, interactive analysis for properties such as safety and deadlocks by utilizing RGA.

It should be noted that "AdaFlow" assumes that the main procedure and all declared tasks activate simultaneously as modeled by the *parbegin* and *parend* control structure. Although not shown in Figure 4.2, execution of a package's sequence of statements or initialization before the *parbegin* has been modeled, but is not reachable. The first code block for a package's sequence of statements is never linked to the rest of the model.

C. ENVIRONMENT

This preliminary work is written in Ada and utilizes the same front-end machine as the automated metric tool "AdaMeasure". "AdaFlow" was originally written and compiled on the Meridian AdaVantage™ Compiler (Compiler Release 2.0). In order to install and operate the AdaVantage compiler, a target system must possess:

- MS-DOS or PC-DOS version 2.1 or later.
- A hard disk (typically 5MB or larger).
- 640K bytes of Random Access Memory in the base memory area.

In addition, an 8087 or 80287 floating point math coprocessor must be installed for programs that use floating point operations. "AdaFlow" currently does not require floating point operations.

AdaVantage is a trademark of Meridian Software Systems, Inc.

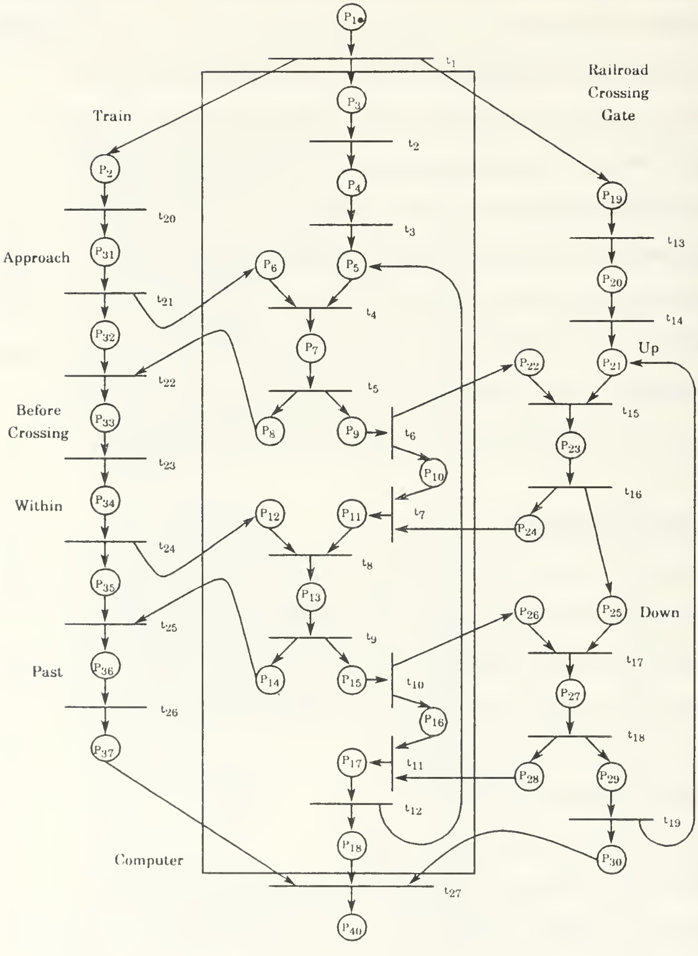


Figure 4.2 An AdaFlow Model of a Simple Railroad Crossing

Release 2.2 of P-NUT is only suitable for systems running a compatible version of 4.2bsd UNIX®. P-NUT was successfully installed at the Naval Postgraduate School on a SUN-3 workstation. To facilitate software analysis in the current form of "AdaFlow", the "AdaFlow" source code was transferred to the SUN workstation and was successfully recompiled using VADS® (Verdix Ada Development System, Version 5.5 for SUN-3) without modification.

All the P-NUT software in release 2.2 is available free of charge from the Information and Computer Science Department of the University of California, Irvine. The point of contact for inquiries concerning P-NUT is Professor Rami Razouk. Release 2.2 includes the C source code and binaries for SUN-3's. If operating in a different 4.2bsd UNIX environment, a Makefile is provided to facilitate recompilation of the source code.

The Ada source code for "AdaFlow" is available free of charge from the Computer Science Department of the Naval Postgraduate School. The point of contact for inquiries concerning "AdaFlow" is LCDR John Yurchak. Supplementary information concerning compilation of the source code is provided along with the source code.

UNIX is a registered trademark of the Bell System
VADS is a registered trademark of the Verdix Corporation

V. CONCLUSION

Ada is the Department of Defense's language of choice for programming embedded, real-time systems. The decision to use Ada has hastened the need for Ada-based, automated software engineering tools. The Petri net-based method proposed by Leveson and Stolzy for analyzing real-time systems has considerable merit; however, hand production of Petri net models for large, complicated systems is a tedious and error-prone process at best. This thesis has described and demonstrated that an efficient method exists for the automated translation of Ada source code to Petri nets. By adding additional features of the Ada language such as separate compilation and a library manager to "AdaFlow", the production and analysis of Petri net models on the systems scale is possible.

A. THE FUTURE

As the primary purpose of this thesis was to describe and demonstrate a methodology for the translation of Ada source code to Petri net models, not all control structures and features of the Ada language have actually been implemented in "AdaFlow"; however, every design decision was made to facilitate the addition of these features. For example, the choice to utilize a scoped symbol table enables one to capitalize on Ada's separate compilation facility at a later date. By adding a library manager to respond to Ada's *with* statement, it is possible to maintain a library of Petri net models. These Petri net models could be of other Ada programs or pre-defined "environment

models" that could be referenced like Ada programs for systems testing of the software. It is envisioned that a library manager would operate by pre-loading the Net Generator with a package's Petri Net model, and the Symbol Table with a package's scoped identifiers and properties.

The Modified Ada Grammar, although able to parse a general Ada program, was developed specifically with metrics in mind. There are a number of ways to massage a grammar to appear LL(1). In their implementation of metrics, Neider and Fairbanks did not have to coordinate searching a scoped symbol table with the grammar. The massaged production rules for NAME reflect this bias. When the same production rules are used while trying to coordinate the search of a scoped symbol table, the grammar becomes hard to read and difficult to use. In "AdaFlow" only simplistic coordination efforts were taken with respect to the production rules for NAME. It was considered more important to demonstrate rather than perfect this capability. As searching the scoped symbol table is necessary to ascertain if an identifier is a procedure call, a function call, or a task entry, the logical candidate for change is the grammar. Future work should include re-massaging this portion of the Modified Ada Grammar to facilitate the coordination of searching a scoped symbol table.

Discussion of analysis of the Petri net models produced by "AdaFlow" has purposely been minimized. For the purpose of this thesis, it is sufficient to note that powerful automated analysis tools such as P-NUT's RGA are currently available. As noted previously, RGA utilizes an input language of first order predicate calculus with the addition of branching-time temporal logic operators. Although this method of interactive analysis is powerful, it

limits the usefulness of the tool to those who have a firm understanding of predicate calculus. Future work on "AdaFlow" should include the design and addition of a high-level, user-friendly interface to this analysis tool. This interface should be able to take user queries and formulate the mathematical expressions understood by RGA.

In the train crossing example presented in Chapter IV, integration of "AdaFlow" software models with environment models was demonstrated by modeling a software test harness. Although this method served to demonstrate the principle of software analysis at the system level, the test harness has limitations in modeling the true environment the software may encounter. In related Petri net research at the Naval Postgraduate School, Lewis (Ref. 16) describes the analysis of a proposed, but never developed, real-time embedded missile software package. This analysis is conducted at the system level using Petri net models of the environment constructed by hand. Further research into using "AdaFlow" to automate the integration of these environment models with the software under analysis is warranted.

It is hoped that as the concept and features of "AdaFlow" are fully developed, this software tool will become a valuable aid in the design and testing of Ada programs for real-time, embedded applications.

APPENDIX A

MODIFIED ADA GRAMMAR

(9.10) (parser3)
ABORT_STATEMENT ⇒ NAME [, NAME]* ;

(9.5) (parser1)
ACCEPT_STATEMENT ⇒ identifier [(EXPRESSION) ?] [FORMAL_PART ?]
[do SEQUENCE_OF_STATEMENTS end [identifier ?] ?] ;

(4.3) (parser3)
AGGREGATE ⇒ (COMPONENT_ASSOCIATION [, COMPONENT_ASSOCIATION]*)

(4.8) (parser3)
ALLOCATOR ⇒ SUBTYPE_INDICATION ['AGGREGATE ?]

(3.6) (parser3)
ARRAY_TYPE_DEFINITION ⇒ (INDEX_CONSTRAINT of SUBTYPE_INDICATION

(5.2) (parser2)
ASSIGNMENT_OR_PROCEDURE_CALL ⇒ NAME : = EXPRESSION ;
⇒ NAME ;

(4.1.4) (parser3)
ATTRIBUTE_DESIGNATOR ⇒ identifier [(EXPRESSION) ?]
⇒ range [(EXPRESSION) ?]
⇒ digits [(EXPRESSION) ?]
⇒ delta [(EXPRESSION) ?]

(3.1) (parser1)
BASIC_DECLARATION ⇒ type TYPE_DECLARATION
⇒ subtype SUBTYPE_DECLARATION
⇒ procedure PROCEDURE_UNIT
⇒ function FUNCTION_UNIT
⇒ package PACKAGE_DECLARATION
⇒ generic GENERIC_DECLARATION
⇒ IDENTIFIER_DECLARATION
⇒ task TASK_DECLARATION

(3.9) (parser1)
BASIC_DECLARATIVE_ITEM ⇒ BASIC_DECLARATIVE
⇒ REPRESENTATION_CLAUSE
⇒ use WITH_OR_USE_CLAUSE

(10.1) (parser0)
BASIC_UNIT ⇒ LIBRARY_UNIT
⇒ SUBUNIT

(4.5) (parser4)
BINARY_ADDING_OPERATOR ⇒ +
⇒ -
⇒ &

(5.6) (parser1)
BLOCK_STATEMENT ⇒ [declare DECLARATIVE_PART ?] begin
SEQUENCE_OF_STATEMENTS [exception
[EXCEPTION_HANDLER]⁺ ?] ?] end [identifier ?] ;

(5.4) (parser1)
CASE_STATEMENT ⇒ EXPRESSION is [CASE_STATEMENT_ALTERNATIVE]⁺ end case ;

(5.4) (parser1)
CASE_STATEMENT_ALTERNATIVE ⇒ when CHOICE [| CHOICE]* = >
SEQUENCE_OF_STATEMENTS

(3.7.3) (parser3)
CHOICE ⇒ EXPRESSION [..SIMPLE_EXPRESSION ?]
⇒ EXPRESSION [CONSTRAINT ?]
⇒ others

(10.1) (parser0)
COMPILATION ⇒ [COMPILATION_UNIT]⁺

(10.1) (parser0)
COMPILATION_UNIT ⇒ CONTEXT_CLAUSE BASIC_UNIT

(4.3) (parser3)
COMPONENT_ASSOCIATION ⇒ [CHOICE [| CHOICE]* = > ?] EXPRESSION

(3.7) (parser2)
COMPONENT_DECLARATION ⇒ IDENTIFIER_LIST : SUBTYPE_INDICATION
[: = EXPRESSION ?] ;

(3.7) (parser2)
COMPONENT_LIST ⇒ [COMPONENT_DECLARATION]* [VARIANT_PART ?]
⇒ null ;

(5.1) (parser1)
COMPOUND_STATEMENT ⇒ if IF_STATEMENT
⇒ case CASE_STATEMENT
⇒ LOOP_STATEMENT
⇒ BLOCK_STATEMENT
⇒ accept ACCEPT_STATEMENT
⇒ SELECT_STATEMENT

(3.2) (parser2)
CONSTANT_TERM ⇒ array ARRAY_TYPE_DEFINITION [: = EXPRESSION ?] ;
⇒ : = EXPRESSION ;

(3.3.2) (parser3)
CONSTRAINT ⇒ range RANGES
⇒ range < >
⇒ digits FLOATING_OR_FIXED_POINT_CONSTRAINT
⇒ delta FLOATING_OR_FIXED_POINT_CONSTRAINT
⇒ (INDEX_CONSTRAINT

(10.1) (parser0)
CONTEXT_CLAUSE ⇒ [with WITH_OR_USE_CLAUSE
[use WITH_OR_USE_CLAUSE]*]*

(3.9) (parser1)
DECLARATIVE_PART ⇒ [BASIC_DECLARATIVE_ITEM]* [LATER_DECLARATIVE_ITEM]*

(9.6) (parser3)
DELAY_STATEMENT ⇒ SIMPLE_EXPRESSION ;

(6.1) (parser2)
DESIGNATOR ⇒ identifier
⇒ string_literal

(3.6) (parser3)
DISCRETE_RANGE ⇒ RANGES [CONSTRAINT ?]

(3.7.1) (parser2)
DISCRIMINANT_PART ⇒ (DISCRIMINANT_SPECIFICATION
[; DISCRIMINANT_SPECIFICATION]*)

(3.7.1) (parser2)
DISCRIMINANT_SPECIFICATION ⇒ IDENTIFIER_LIST : NAME [: = EXPRESSION ?]

(9.5) (parser2)
ENTRY_DECLARATION ⇒ entry identifier [(DISCRETE_RANGE) ?]
[FORMAL_PART ?] ;

(3.5.1) (parser4)
ENUMERATION_LITERAL ⇒ identifier
⇒ character_literal

(3.5.1) (parser4)
ENUMERATION_TYPE_DEFINITION ⇒ (ENUMERATION_LITERAL
[, ENUMERATION_LITERAL]*)

(11.1) (parser2)
EXCEPTION_CHOICE ⇒ NAME
⇒ others

(11.2) (parser1)
EXCEPTION_HANDLER ⇒ when EXCEPTION_CHOICE [| EXCEPTION_CHOICE]*
= >SEQUENCE_OF_STATEMENTS

(8.5) (parser2)
EXCEPTION_TAIL ⇒ ;
⇒ renames NAME ;

(5.7) (parser3)
EXIT_STATEMENT ⇒ [NAME ?] [when EXPRESSION ?] ;

(4.4) (parser3)
EXPRESSION ⇒ RELATION [RELATION_TAIL ?]

(4.4) (parser3)
FACTOR ⇒ PRIMARY [** PRIMARY ?]
⇒ abs PRIMARY
⇒ not PRIMARY

(3.5.7) (parser3)
FLOATING_OR_FIXED_POINT_CONSTRAINT ⇒ SIMPLE_EXPRESSION [range RANGES ?]

(6.4) (parser4)
FORMAL_PARAMETER ⇒ identifier = >

(6.1) (parser2)
FORMAL_PART ⇒(PARAMETER_SPECIFICATION [; PARAMETER_SPECIFICATION]*)

(6.1) (parser1)
FUNCTION_UNIT ⇒ DESIGNATOR [FORMAL_PART ?] return NAME is
SUBPROGRAM_BODY
⇒ DESIGNATOR [FORMAL_PART ?] return NAME ;
⇒ DESIGNATOR [FORMAL_PART ?] return NAME renames NAME ;
⇒ DESIGNATOR is SUBPROGRAM_BODY

(12.1) (parser2)
GENERIC_ACTUAL_PART ⇒ (GENERIC_ASSOCIATION [, GENERIC_ASSOCIATION]*)

(12.1) (parser2)
GENERIC_ASSOCIATION ⇒ [GENERIC_FORMAL_PARAMETER ?] EXPRESSION

(12.1) (parser1)
GENERIC_DECLARATION ⇒ [GENERIC_PARAMETER_DECLARATION]*
GENERIC_FORMAL_PART

(12.1) (parser2)
GENERIC_FORMAL_PARAMETER ⇒ identifier = >
⇒ string_literal = >

(12.1) (parser1)
```
GENERIC_FORMAL_PART     ⇒ procedure PROCEDURE_UNIT
                        ⇒ function FUNCTION_UNIT
                        ⇒ package PACKAGE_DECLARATION
```

(12.1) (parser1)
```
GENERIC_PARAMETER_DECLARATION  ⇒ IDENTIFIER_LIST : [MODE ?] NAME
                                      [: = EXPRESSION ?] ;
                               ⇒ type private [DISCRIMINANT_PART ?]
                                      is PRIVATE_TYPE_DECLARATION ;
                               ⇒ type private [DISCRIMINANT_PART ?]
                                      is GENERIC_TYPE_DEFINITION ;
                               ⇒ with procedure PROCEDURE_UNIT
                               ⇒ with function FUNCTION_UNIT
```

(12.1) (parser2)
```
GENERIC_TYPE_DEFINITION ⇒ ( < > )
                        ⇒ range < >
                        ⇒ digits < >
                        ⇒ delta < >
                        ⇒ array ARRAY_TYPE_DEFINITION
                        ⇒ access SUBTYPE_INDICATION
```

(5.9) (parser3)
```
GOTO_STATEMENT ⇒     NAME ;
```

(3.2) (parser2)
```
IDENTIFIER_DECLARATION ⇒IDENTIFIER_LIST : IDENTIFIER_DECLARATION_TAIL
```

(3.2) (parser2)
```
IDENTIFIER_DECLARATION_TAIL ⇒ exception EXCEPTION_TAIL
                            ⇒ constant CONSTANT_TERM
                            ⇒ array ARRAY_TYPE_DEFINITION
                                   [: = EXPRESSION ?] ;
                            ⇒ NAME  IDENTIFIER_TAIL
```

(3.2) (parser2)
```
IDENTIFIER_LIST ⇒    identifier [, identifier]*
```

(3.2) (parser2)
```
IDENTIFIER_TAIL  ⇒ [CONSTRAINT ?] [: = EXPRESSION ?] ;
                 ⇒ [renames NAME ?] ;
```

(5.3) (parser1)
```
IF_STATEMENT     ⇒ EXPRESSION then SEQUENCE_OF_STATEMENTS
                       [elsif EXPRESSION then SEQUENCE_OF_STATEMENTS]* [else
                       SEQUENCE_OF_STATEMENTS ?] end if ;
```

(3.6) (parser3)
```
INDEX_CONSTRAINT ⇒    DISCRETE_RANGE [, DISCRETE_RANGE]* )
```

(3.5.4) (parser3)
INTEGER_TYPE_DEFINITION ⇒ range RANGES

(5.5) (parser3)
ITERATION_SCHEME ⇒ while EXPRESSION
⇒ for LOOP_PARAMETER_SPECIFICATION

(5.1) (parser2)
LABEL ⇒ << identifier >>

(3.9) (parser1)
LATER_DECLARATIVE_ITEM ⇒ PROPER_BODY
⇒ generic GENERIC_DECLARATION
⇒ use WITH_OR_USE_CLAUSE

(4.1) (parser3)
LEFT_PAREN_NAME_TAIL ⇒ [FORMAL_PARAMETER ?] EXPRESSION [..EXPRESSION ?]
[, [FORMAL_PARAMETER ?] EXPRESSION
[..EXPRESSION ?]]*) [NAME_TAIL]*

(10.1) (parser0)
LIBRARY_UNIT ⇒ procedure PROCEDURE_UNIT
⇒ function FUNCTION_UNIT
⇒ package PACKAGE_DECLARATION
⇒ generic GENERIC_DECLARATION

(5.5) (parser3)
LOOP_PARAMETER_SPECIFICATION ⇒ identifier in [reverse ?] DISCRETE_RANGE

(5.5) (parser1)
LOOP_STATEMENT ⇒ [ITERATION_SCHEME ?] loop
SEQUENCE_OF_STATEMENTS end loop [identifier ?] ;

(6.1) (parser2)
MODE ⇒ [in ?]
⇒ in out
⇒ out

(4.5) (parser4)
MULTIPLYING_OPERATOR ⇒ *
⇒ /
⇒ mod
⇒ rem

(4.1) (parser3)
NAME ⇒ identifier [NAME_TAIL ?]
⇒ character_literal [NAME_TAIL ?]
⇒ string_literal [NAME_TAIL ?]

(4.1) (parser3)
NAME_TAIL ⇒ (LEFT_PAREN_NAME_TAIL
⇒ .SELECTOR [NAME_TAIL]*
⇒ 'AGGREGATE [NAME_TAIL]*
⇒ 'ATTRIBUTE_DESIGNATOR [NAME_TAIL]*

(7.1) (parser1)
PACKAGE_DECLARATION ⇒ body identifier is SUBPROGRAM_BODY
⇒ identifier is PACKAGE_TAIL_END
⇒ identifier renames NAME;

(7.1) (parser1)
PACKAGE_TAIL_END ⇒ new NAME [GENERIC_ACTUAL_PART ?] ;
⇒ [BASIC_DECLARATIVE_ITEM]* [private
[BASIC_DECLARATIVE_ITEM]* ?] end [identifier ?] ;

(6.1) (parser2)
PARAMETER_SPECIFICATION ⇒ IDENTIFIER_LIST : MODE NAME [: = EXPRESSION ?]

(4.4) (parser3)
PRIMARY ⇒ numeric_literal
⇒ null
⇒ string_literal
⇒ new ALLOCATOR
⇒ NAME
⇒ AGGREGATE

(7.4) (parser2)
PRIVATE_TYPE_DECLARATION ⇒ [limited ?] private

(6.1) (parser1)
PROCEDURE_UNIT ⇒ identifier [FORMAL_PART ?] is SUBPROGRAM_BODY
⇒ identifier [FORMAL_PART ?] ;
⇒ identifier [FORMAL_PART ?] renames NAME ;

(3.9) (parser1)
PROPER_BODY ⇒ procedure PROCEDURE_UNIT
⇒ function FUNCTION_UNIT
⇒ package PACKAGE_DECLARATION
⇒ task TASK_DECLARATION

(3.5) (parser3)
RANGES ⇒ SIMPLE_EXPRESSION [..SIMPLE_EXPRESSION ?]

(11.3) (parser3)
RAISE_STATEMENT ⇒ [NAME ?] ;

(13.4) (parser2)
RECORD_REPRESENTATION_CLAUSE ⇒ [at mod SIMPLE_EXPRESSION ?]
[NAME at SIMPLE_EXPRESSION range
RANGES]* end record ;

(3.7) (parser2)
RECORD_TYPE_DEFINITION ⇒ COMPONENT_LIST end record

(4.4) (parser3)
RELATION ⇒ SIMPLE_EXPRESSION [SIMPLE_EXPRESSION_TAIL ?]

(4.4) (parser3)
RELATION_TAIL ⇒ [and [then ?] RELATION]*
⇒ [or [else ?] RELATION]*
⇒ [xor RELATION]*

(4.5) (parser4)
RELATIONAL_OPERATOR ⇒ =
⇒ /=
⇒ <
⇒ < =
⇒ >
⇒ > =

(13.1) (parser2)
REPRESENTATION_CLAUSE ⇒ for NAME use record
RECORD_REPRESENTATION_CLAUSE
⇒ for NAME use [at ?] SIMPLE_EXPRESSION;

(5.8) (parser3)
RETURN_STATEMENT ⇒ [EXPRESSION ?] ;

(9.7.1) (parser1)
SELECT_ALTERNATIVE ⇒ [when EXPRESSION = > ?] accept ACCEPT_STATEMENT
[SEQUENCE_OF_STATEMENTS ?]
⇒ [when EXPRESSION = > ?] delay DELAY_STATEMENT
[SEQUENCE_OF_STATEMENTS ?]
⇒ [when EXPRESSION = > ?] terminate ;

(9.7.1) (parser1)
SELECT_ENTRY_CALL ⇒ else SEQUENCE_OF_STATEMENTS
⇒ or delay DELAY_STATEMENT
[SEQUENCE_OF_STATEMENTS ?]

(9.7) (parser1)
SELECT_STATEMENT ⇒ select SELECT_STATEMENT_TAIL [SELECT_ENTRY_CALL ?]
end select ;

(9.7.1) (parser1)
SELECT_STATEMENT_TAIL ⇒ SELECT_ALTERNATIVE [or SELECT_ALTERNATIVE]*
⇒ NAME ; [SEQUENCE_OF_STATEMENTS ?]

(4.1.3) (parser4)
SELECTOR ⇒ identifier
⇒ character_literal

⇒ string_literal
⇒ all

(5.1) (parser1)
SEQUENCE_OF_STATEMENTS ⇒ [STATEMENT]+

(4.4) (parser3)
SIMPLE_EXPRESSION ⇒ [+ ?] TERM [BINARY_ADDING_OPERATOR TERM]*
⇒ [- ?] TERM [BINARY ADDING_OPERATOR TERM]*

(4.4) (parser3)
SIMPLE_EXPRESSION_TAIL ⇒ RELATIONAL_OPERATOR SIMPLE_EXPRESSION
⇒ [not ?] in RANGES
⇒ [not ?] in NAME

(5.1) (parser2)
SIMPLE_STATEMENT ⇒ null ;
⇒ ASSIGNMENT_OR_PROCEDURE_CALL
⇒ exit EXIT_STATEMENT
⇒ return RETURN_STATEMENT
⇒ goto GOTO_STATEMENT
⇒ delay DELAY_STATEMENT
⇒ abort ABORT_STATEMENT
⇒ raise RAISE_STATEMENT

(5.1) (parser1)
STATEMENT ⇒ [LABEL ?] SIMPLE_STATEMENT
⇒ [LABEL ?] COMPOUND_STATEMENT

(6.3) (parser1)
SUBPROGRAM_BODY ⇒ new NAME [GENERIC_ACTUAL_PART ?] ;
⇒ separate ;
⇒ <> ;
⇒ [DECLARATIVE_PART ?] [begin SEQUENCE_OF_STATEMENTS [exception [EXCEPTION_HANDLER]+ ?]?] end [DESIGNATOR ?] ;
⇒ NAME ;

(3.3.2) (parser2)
SUBTYPE_DECLARATION ⇒ identifier is SUBTYPE_INDICATION ;

(3.3.2) (parser3)
SUBTYPE_INDICATION ⇒ NAME [CONSTRAINT ?]

(10.1) (parser0)
SUBUNIT ⇒ separate (NAME) PROPER_BODY

(9.1) (parser1)
TASK_DECLARATION ⇒ body identifier is SUBPROGRAM_BODY ;
⇒ [type ?] identifier [is [ENTRY_DECLARATION]* [REPRESENTATION_CLAUSE]* end [identifier ?] ?] ;

(4.4) (parser3)
TERM ⇒ FACTOR [MULTIPLYING_OPERATOR FACTOR]*

(3.3.1) (parser2)
TYPE_DECLARATION ⇒ identifier [DISCRIMINANT_PART ?]
[is PRIVATE_TYPE_DECLARATION ?] ;
⇒ identifier [DISCRIMINANT_PART ?]
[is TYPE_DEFINITION ?] ;

(3.3.1) (parser2)
TYPE_DEFINITION ⇒ ENUMERATION_TYPE_DEFINITION
⇒ INTEGER_TYPE_DEFINITION
⇒ digits FLOATING_OR_FIXED_POINT_CONSTRAINT
⇒ delta FLOATING_OR_FIXED_POINT_CONSTRAINT
⇒ array ARRAY_TYPE_DEFINITION
⇒ record RECORD_TYPE_DEFINITION
⇒ access SUBTYPE_INDICATION
⇒ new SUBTYPE_INDICATION

(3.7.3) (parser2)
VARIANT ⇒ when CHOICE [| CHOICE]* => COMPONENT_LIST

(3.7.3) (parser2)
VARIANT_PART ⇒ case identifier is [VARIANT]* end case ;

(10.1.1) (parser2)
WITH_OR_USE_CLAUSE ⇒ identifier [, identifier]* ;

APPENDIX B

"ADAFLOW" PROGRAM LISTING - MAIN

```
--***************************************************************--
--                                                              --
--  TITLE:          ADAFLOW                                     --
--                                                              --
--  MODULE NAME:    PROCEDURE MAIN                              --
--  FILE NAME:      MAIN.ADA                                    --
--                                                              --
--  DATE CREATED:   02 FEB 88                                   --
--  LAST MODIFIED:  28 APR 88                                   --
--                                                              --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                    --
--                                                              --
--  DESCRIPTION:  This procedure is the highest level procedure --
--                of ADAFLOW.  It queries the user for an ADA   --
--                program to model, sets up the token matcher,  --
--                starts the parser through the ADA program, and --
--                translates the results of the parse to P-NUT  --
--                code.                                         --
--                                                              --
--***************************************************************--

with  TOKEN_MATCHER, CODE_BLOCKER, SYMBOL_TABLE,
      NET_GENERATOR, PARSER, TEXT_IO;

procedure MAIN is
  SOURCE_CODE_FILE : string (1..80) := (others => ' ');
  SOURCE_CODE_FILE_LENGTH : natural;

  procedure GET_FILE_NAME is
  UNKNOWN_NAME : exception;
  use TEXT_IO;
  begin
    put_line("WELCOME TO ADAFLOW"); new_line;
    put_line("ENTER THE NAME OF AN ADA SOURCE FILE TO MODEL"); new_line;
    SOURCE_CODE_FILE := (others => ' ');
    get_line(SOURCE_CODE_FILE, SOURCE_CODE_FILE_LENGTH); new_line;
    if (SOURCE_CODE_FILE_LENGTH = 0) then
      raise UNKNOWN_NAME;
    else
      put_line(SOURCE_CODE_FILE(1..SOURCE_CODE_FILE_LENGTH));
    end if;
  end GET_FILE_NAME;
begin
```

```
  GET_FILE_NAME;
  TOKEN_MATCHER.SET_UP_TOKEN_MATCHER(SOURCE_CODE_FILE(1..
                                                   SOURCE_CODE_FILE_LENGTH));
  TEXT_IO.put_line("PARSING BEGINS . . . ");
  if PARSER.IS_PARSED then
    TEXT_IO.put_line(". . . PARSE SUCCESSFUL");
    NET_GENERATOR.TRANSLATE_TO_PEANUT;
  else
    TEXT_IO.put_line(". . . PARSE UNSUCCESSFUL");
    CODE_BLOCKER.CLEAR_CODE_BLOCKER;
    NET_GENERATOR.RESET_NET_GENERATOR;
  end if;
  SYMBOL_TABLE.CLEAR_SYM_TAB;
  TOKEN_MATCHER.RELEASE_TOKEN_MATCHER;
exception
  when others =>
    TEXT_IO.put_line("UNABLE TO MODEL ADA SOURCE CODE");
    TEXT_IO.put_line(". . . PARSE UNSUCCESSFUL");
    CODE_BLOCKER.CLEAR_CODE_BLOCKER;
    NET_GENERATOR.RESET_NET_GENERATOR;
    SYMBOL_TABLE.CLEAR_SYM_TAB;
    begin
      TOKEN_MATCHER.RELEASE_TOKEN_MATCHER;
    exception
      when others => null;
    end;
end MAIN;
```

APPENDIX C

"ADAFLOW" PROGRAM LISTING - PARSER

```
--****************************************************************--
--                                                                  --
--   TITLE:          ADAFLOW                                        --
--                                                                  --
--   MODULE NAME:    PACKAGE PARSER                                 --
--   FILE NAME:      PARSER.ADS                                     --
--                                                                  --
--   DATE CREATED:   18 FEB 88                                      --
--   LAST MODIFIED:  28 APR 88                                      --
--                                                                  --
--   AUTHOR(S):      LT ALBERT J. GRECCO, USN                       --
--                                                                  --
--   DESCRIPTION:  This package defines the only interfaces to     --
--        to the parser.  Packages PARSER_0 through PARSER_4        --
--        exist only as local packages to package PARSER and are   --
--        not user accessable.                                      --
--                                                                  --
--****************************************************************--

package PARSER is
  function IS_PARSED return boolean;
  -- pre  - TOKEN_MATCHER, SYMBOL_TABLE, CODE_BLOCKER, and NET_GENERATOR are
  --        initialized.
  -- post - If the file being parsed is a valid ADA program, IS_PARSED
  --        is TRUE else IS_PARSED is FALSE.
end PARSER;
```

```
--*****************************************************************--
--                                                                 --
--  TITLE:          ADAFLOW                                        --
--                                                                 --
--  MODULE NAME:    PACKAGE PARSER                                 --
--  FILE NAME:      PARSER.ADB                                     --
--                                                                 --
--  DATE CREATED:   18 FEB 88                                      --
--  LAST MODIFIED:  28 APR 88                                      --
--                                                                 --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                       --
--                                                                 --
--  DESCRIPTION:  This package implements the only interfaces to --
--       the parser.                                               --
--                                                                 --
--*****************************************************************--

with PARSER_0, PARSER_4;

package body PARSER is
  function IS_PARSED return boolean is
  -- pre  - TOKEN_MATCHER, SYMBOL_TABLE, CODE_BLOCKER, and NET_GENERATOR have
  --        been initialized.
  -- post - If the file being parsed is a valid ADA program, IS_PARSED
  --        is TRUE else IS_PARSED is FALSE.
  begin
    return PARSER_0.COMPILATION;
  exception
    when PARSER_4.PARSER_ERROR =>
      return FALSE;
    when others =>
      raise;
    end IS_PARSED;
end PARSER;
```

```
--**************************************************************--
--                                                              --
--  TITLE:          ADAFLOW                                     --
--                                                              --
--  MODULE NAME:    PACKAGE PARSER_0                            --
--  FILE NAME:      PARSER0.ADS                                 --
--                                                              --
--  DATE CREATED:   18 FEB 88                                   --
--  LAST MODIFIED:  28 APR 88                                   --
--                                                              --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                    --
--                                                              --
--  BASED ON A MODIFIED ADA GRAMMAR DEVELOPED BY:               --
--                  LCDR JEFFREY L. NIEDER, USN                 --
--                  LT KARL S. FAIRBANKS, JR., USN              --
--                  LCDR PAUL M. HERZIG, USN                    --
--                                                              --
--  DESCRIPTION:  This package defines the functions that       --
--       make up the highest level productions for a top-down,  --
--       recursive descent parser.                              --
--                                                              --
--**************************************************************--

package PARSER_0 is
  function COMPILATION return boolean;
  function COMPILATION_UNIT return boolean;
  function CONTEXT_CLAUSE return boolean;
  function BASIC_UNIT return boolean;
  function LIBRARY_UNIT return boolean;
  function SUBUNIT return boolean;
end PARSER_0;
```

```
--****************************************************************--
--                                                                --
--  TITLE:          ADAFLOW                                       --
--                                                                --
--  MODULE NAME:    PACKAGE PARSER_0                              --
--  FILE NAME:      PARSER0.ADB                                   --
--                                                                --
--  DATE CREATED:   18 FEB 88                                     --
--  LAST MODIFIED:  28 APR 88                                     --
--                                                                --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                      --
--                                                                --
--  BASED ON A MODIFIED ADA GRAMMAR DEVELOPED BY:                 --
--                  LCDR JEFFREY L. NIEDER, USN                   --
--                  LT KARL S. FAIRBANKS, JR., USN                --
--                  LCDR PAUL M. HERZIG, USN                      --
--                                                                --
--  DESCRIPTION:  This package implements the functions that      --
--       make up the highest level productions for a top-down,    --
--       recursive descent parser.  Each function is preceded     --
--       by the grammar productions they are implementing.        --
--                                                                --
--****************************************************************--

with PARSER_1, PARSER_2, PARSER_3, PARSER_4, TOKEN_MATCHER;

package body PARSER_0 is
  package TM renames TOKEN_MATCHER;
  package P1 renames PARSER_1;
  package P2 renames PARSER_2;
  package P3 renames PARSER_3;
  package P4 renames PARSER_4;

  -- COMPILATION -->  [COMPILATION_UNIT]+
function COMPILATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("COMPILATION");
  end if;
  if (COMPILATION_UNIT) then
    while (COMPILATION_UNIT) loop
      null;
    end loop;
    return (TRUE);
  else
    return (FALSE);
  end if;
end COMPILATION;

----------------------------------------------------------------------------
```

```
  -- COMPILATION_UNIT -->  CONTEXT_CLAUSE  BASIC_UNIT
function COMPILATION_UNIT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("COMPILATION_UNIT");
  end if;
  if (CONTEXT_CLAUSE) then
     if (BASIC_UNIT) then
        return (TRUE);
     else
        return (FALSE);
     end if;
  else
    return (FALSE);
  end if;
end COMPILATION_UNIT;

-------------------------------------------------------------------------------

  -- CONTEXT_CLAUSE -->  [with WITH_OR_USE_CLAUSE [use WITH_OR_USE_CLAUSE]* ]*
function CONTEXT_CLAUSE return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("CONTEXT_CLAUSE");
  end if;
  while (TM.MATCH(TM.TOKEN_WITH)) loop
    if not (P2.WITH_OR_USE_CLAUSE) then
      P4.SYNTAX_ERROR("Context clause");
    end if;
    while (TM.MATCH(TM.TOKEN_USE)) loop
      if not (P2.WITH_OR_USE_CLAUSE) then
        P4.SYNTAX_ERROR("Context clause");
      end if;
    end loop;      -- inner while loop
  end loop;        -- outer while loop
  return (TRUE);
end CONTEXT_CLAUSE;

-------------------------------------------------------------------------------

  -- BASIC_UNIT -->  LIBRARY_UNIT
  --            -->  SUBUNIT
function BASIC_UNIT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("BASIC_UNIT");
  end if;
  if (LIBRARY_UNIT) then
    return (TRUE);
  elsif (SUBUNIT) then
    return (TRUE);
```

```
  else
    return (FALSE);
  end if;
end BASIC_UNIT;

--------------------------------------------------------------------------------

  -- LIBRARY_UNIT -->  procedure PROCEDURE_UNIT
  --              -->  function FUNCTION_UNIT
  --              -->  package PACKAGE_DECLARATION
  --              -->  generic GENERIC_DECLARATION
function LIBRARY_UNIT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("LIBRARY_UNIT");
  end if;
  if (TM.MATCH(TM.TOKEN_PROCEDURE)) then
    if (P1.PROCEDURE_UNIT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Library unit");
    end if;  -- if procedure_unit statement
  elsif (TM.MATCH(TM.TOKEN_FUNCTION)) then
    if (P1.FUNCTION_UNIT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Library unit");
    end if;  -- if function_unit statement
  elsif (TM.MATCH(TM.TOKEN_PACKAGE)) then
    if (P1.PACKAGE_DECLARATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Library unit");
    end if;  -- if package_declaration
  elsif (TM.MATCH(TM.TOKEN_GENERIC)) then
    if (P1.GENERIC_DECLARATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Library unit");
    end if;  -- if generic_declaration
  else
    return (FALSE);
  end if;
end LIBRARY_UNIT;

--------------------------------------------------------------------------------

  -- SUBUNIT -->  separate (NAME) PROPER_BODY
function SUBUNIT return boolean is
begin
  if (P4.PRINT_CALLS) then
```

```
    P4.OUT_PUT("SUBUNIT");
  end if;
  if (TM.MATCH(TM.TOKEN_SEPARATE)) then
    if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
      if (P3.NAME) then
        if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
          if (P1.PROPER_BODY) then
            return (TRUE);
          else
            P4.SYNTAX_ERROR("Subunit");
          end if;  -- if proper_body statement
        else
          P4.SYNTAX_ERROR("Subunit");
        end if;  -- if bypass(token_right_paren)
      else
        P4.SYNTAX_ERROR("Subunit");
      end if;  -- if name statement
    else
      P4.SYNTAX_ERROR("Subunit");
    end if;  -- if bypass(token_left_paren)
  else
    return (FALSE);
  end if;  -- if bypass(token_separate)
end SUBUNIT;

end PARSER_0;
```

```
--*************************************************************--
--                                                             --
--  TITLE:          ADAFLOW                                    --
--                                                             --
--  MODULE NAME:    PACKAGE PARSER_1                           --
--  FILE NAME:      PARSER1.ADS                                --
--                                                             --
--  DATE CREATED:   18 FEB 88                                  --
--  LAST MODIFIED:  28 APR 88                                  --
--                                                             --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                   --
--                                                             --
--  BASED ON A MODIFIED ADA GRAMMAR DEVELOPED BY:              --
--                  LCDR JEFFREY L. NIEDER, USN                --
--                  LT KARL S. FAIRBANKS, JR., USN             --
--                  LCDR PAUL M. HERZIG, USN                   --
--                                                             --
--  DESCRIPTION:  This package defines the functions           --
--       that make up the top level productions for a top-down, --
--       recursive descent parser.                             --
--                                                             --
--*************************************************************--

package PARSER_1 is
  function GENERIC_DECLARATION return boolean;
  function GENERIC_PARAMETER_DECLARATION return boolean;
  function GENERIC_FORMAL_PART return boolean;
  function PROCEDURE_UNIT return boolean;
  function SUBPROGRAM_BODY return boolean;
  function FUNCTION_UNIT return boolean;
  function TASK_DECLARATION return boolean;
  function PACKAGE_DECLARATION return boolean;
  function PACKAGE_TAIL_END return boolean;
  function DECLARATIVE_PART return boolean;
  function BASIC_DECLARATIVE_ITEM return boolean;
  function BASIC_DECLARATION return boolean;
  function LATER_DECLARATIVE_ITEM return boolean;
  function PROPER_BODY return boolean;
  function SEQUENCE_OF_STATEMENTS return boolean;
  function STATEMENT return boolean;
  function COMPOUND_STATEMENT return boolean;
  function BLOCK_STATEMENT return boolean;
  function IF_STATEMENT return boolean;
  function CASE_STATEMENT return boolean;
  function CASE_STATEMENT_ALTERNATIVE return boolean;
  function LOOP_STATEMENT return boolean;
  function EXCEPTION_HANDLER return boolean;
  function ACCEPT_STATEMENT return boolean;
  function SELECT_STATEMENT return boolean;
  function SELECT_STATEMENT_TAIL return boolean;
  function SELECT_ALTERNATIVE return boolean;
```

```
  function SELECT_ENTRY_CALL return boolean;
end PARSER_1;
```

```
--******************************************************************--
--                                                                   --
--  TITLE:          ADAFLOW                                          --
--                                                                   --
--  MODULE NAME:    PACKAGE PARSER_1                                 --
--  FILE NAME:      PARSER1.ADB                                      --
--                                                                   --
--  DATE CREATED:   18 FEB 88                                        --
--  LAST MODIFIED:  28 APR 88                                        --
--                                                                   --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                         --
--                                                                   --
--  BASED ON A MODIFIED ADA GRAMMAR DEVELOPED BY:                    --
--                  LCDR JEFFREY L. NIEDER, USN                      --
--                  LT KARL S. FAIRBANKS, JR., USN                   --
--                  LCDR PAUL M. HERZIG, USN                         --
--                                                                   --
--  DESCRIPTION:  This package implements the functions              --
--       that make up the top level productions for a top-down,      --
--       recursive descent parser. Each function is preceded         --
--       by the grammar productions they are implementing.           --
--                                                                   --
--******************************************************************--

with PARSER_2, PARSER_3, PARSER_4,
     TOKEN_MATCHER, TOKEN_SCANNER, CODE_BLOCKER,
     SYMBOL_TABLE, NET_GENERATOR;

package body PARSER_1 is
  package TM renames TOKEN_MATCHER;
  package P2 renames PARSER_2;
  package P3 renames PARSER_3;
  package P4 renames PARSER_4;

  IS_MAIN_PROGRAM : boolean := TRUE;

  -- GENERIC_DECLARATION -->  [GENERIC_PARAMETER_DECLARATION ]*
  --                           GENERIC_FORMAL_PART
function GENERIC_DECLARATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("GENERIC_DECLARATION");
  end if;
  while (GENERIC_PARAMETER_DECLARATION) loop
    null;
  end loop;
  if (GENERIC_FORMAL_PART) then
    return(TRUE);
  else
    return (FALSE);
```

```
  end if;
end GENERIC_DECLARATION;

--------------------------------------------------------------------------------

  -- GENERIC_PARAMETER_DECLARATION -->  IDENTIFIER_LIST : [MODE ?] NAME
  --                                      [:= EXPRESSION ?] ;
  --                                -->  type private [DISCRIMINANT_PART ?]
  --                                      is PRIVATE_TYPE_DECLARATION ;
  --                                -->  type private [DISCRIMINANT_PART ?]
  --                                      is GENERIC_TYPE_DEFINITION ;
  --                                -->  with procedure PROCEDURE_UNIT
  --                                -->  with function FUNCTION_UNIT
function GENERIC_PARAMETER_DECLARATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("GENERIC_PARAMETER_DECLARATION");
  end if;
  if (P2.IDENTIFIER_LIST) then
    if (TM.MATCH(TM.TOKEN_COLON)) then
      if (P2.MODE) then
        null;
      end if;                               -- if mode statement
      if (P3.NAME) then  -- check for type_mark
        if (TM.MATCH(TM.TOKEN_ASSIGNMENT)) then
          if (P3.EXPRESSION) then
            null;
          else
            P4.SYNTAX_ERROR("Generic parameter declaration");
          end if;                           -- if expression statement
        end if;                             -- if match(token_assignment)
        if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Generic parameter declaration");
        end if;                             -- if match(token_semicolon)
      else
        P4.SYNTAX_ERROR("Generic parameter declaration");
      end if;                               -- if type_mark statement
    else
      P4.SYNTAX_ERROR("Generic parameter declaration");
    end if;                                 -- if match(token_colon)
  elsif (TM.MATCH(TM.TOKEN_TYPE)) then
    if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
      if (P2.DISCRIMINANT_PART) then
        null;
      end if;                               -- if discriminant_part
      if (TM.MATCH(TM.TOKEN_IS)) then
        if (P2.PRIVATE_TYPE_DECLARATION) then
          if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
            return (TRUE);
```

```
        else
          P4.SYNTAX_ERROR("Generic parameter declaration");
        end if;                           -- if match(token_semicolon)
      elsif (P2.GENERIC_TYPE_DEFINITION) then
        if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Generic parameter declaration");
        end if;                           -- if match(token_semicolon)
      else
       P4.SYNTAX_ERROR("Generic parameter declaration");
      end if;                             -- if private_type_declaration
    else
      P4.SYNTAX_ERROR("Generic parameter declaration");
    end if;                               -- if match(token_is)
  else
    P4.SYNTAX_ERROR("Generic parameter declaration");
  end if;                                 -- if match(token_identifier)
elsif (TM.MATCH(TM.TOKEN_WITH)) then
  if (TM.MATCH(TM.TOKEN_PROCEDURE)) then
    if (PROCEDURE_UNIT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Generic parameter declaration");
    end if;                               -- if procedure_unit statement
  elsif (TM.MATCH(TM.TOKEN_FUNCTION)) then
    if (FUNCTION_UNIT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Generic parameter declaration");
    end if;                               -- if function_unit statement
  else
    P4.SYNTAX_ERROR("Generic parameter declaration");
  end if;                                 -- if match(token_procedure)
else
  return (FALSE);
end if;                                   -- if identifier_list
end GENERIC_PARAMETER_DECLARATION;

--------------------------------------------------------------------------------

  -- GENERIC_FORMAL_PART -->  procedure PROCEDURE_UNIT
  --                     -->  function FUNCTION_UNIT
  --                     -->  package PACKAGE_DECLARATION
function GENERIC_FORMAL_PART return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("GENERIC_FORMAL_PART");
  end if;
  if (TM.MATCH(TM.TOKEN_PROCEDURE)) then
    if (PROCEDURE_UNIT) then
```

```
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Generic formal part");
    end if;                                    -- if procedure_unit statement
  elsif (TM.MATCH(TM.TOKEN_FUNCTION)) then
    if (FUNCTION_UNIT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Generic formal part");
    end if;                                    -- if function_unit statement
  elsif (TM.MATCH(TM.TOKEN_PACKAGE)) then
    if (PACKAGE_DECLARATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Generic formal part");
    end if;                                    -- if package_declaration
  else
    return (FALSE);
  end if;
end GENERIC_FORMAL_PART;

--------------------------------------------------------------------------------

  -- PROCEDURE_UNIT -->  identifier [FORMAL_PART ?] is SUBPROGRAM_BODY
  --                -->  identifier [FORMAL_PART ?] ;
  --                -->  identifier [FORMAL_PART ?] renames NAME ;
function PROCEDURE_UNIT return boolean is
START_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION : natural;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("PROCEDURE_UNIT");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    TM.MATCHED_TOKEN(START_TOKEN);
    CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "PROCEDURE CODE BLOCK");
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    LOCATION := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.LEXEME(1..START_TOKEN.LEXEME_SIZE),
                                SYMBOL_TABLE.PROCEDURE_DECLARATION_TAG,
                                LOCATION);
    SYMBOL_TABLE.INSERT_SYM_TAB("END", SYMBOL_TABLE.LABEL_NAME, 0);
    if (IS_MAIN_PROGRAM) then
      NET_GENERATOR.START(SYMBOL_TABLE.FIND_KEY(START_TOKEN.LEXEME(1..
                                                   START_TOKEN.LEXEME_SIZE)));
      IS_MAIN_PROGRAM := FALSE;
    end if;
    if (P2.FORMAL_PART) then
      null;
    end if;                                    -- if formal part statement
    if (TM.MATCH(TM.TOKEN_IS)) then
```

```
      if (SUBPROGRAM_BODY)  then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Procedure unit");
      end if;                                 -- if subprogram body statement
    elsif (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
      SYMBOL_TABLE.EXIT_SCOPE;
      SYMBOL_TABLE.UPDATE_SYM_TAB(0);
      return (TRUE);
    elsif (TM.MATCH(TM.TOKEN_RENAMES)) then
      CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
      SYMBOL_TABLE.EXIT_SCOPE;
      SYMBOL_TABLE.UPDATE_SYM_TAB(0);
      if (P3.NAME) then
        if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Procedure unit");
        end if;                               -- if match(token_semicolon)
      else
        P4.SYNTAX_ERROR("Procedure unit");
      end if;                                 -- if name statement
    end if;                                   -- if match(token_is)
  else
    return (FALSE);
  end if;                                     -- if match(token_identifier)
end PROCEDURE_UNIT;

-------------------------------------------------------------------------------

  -- SUBPROGRAM_BODY -->  new NAME [GENERIC_ACTUAL_PART ?] ;
  --                 -->  separate ;
  --                 -->  <> ;
  --                 -->  [DECLARATIVE_PART ?] [begin SEQUENCE_OF_STATEMENTS
  --                [exception [EXCEPTION_HANDLER]+ ?]?] end [DESIGNATOR ?] ;
  --                 -->  NAME ;
function SUBPROGRAM_BODY return boolean is
START_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
STOP_TOKEN  : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE : natural;
LOCATION_TWO : natural;
use SYMBOL_TABLE;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SUBPROGRAM_BODY");
  end if;
  LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  if (TM.MATCH(TM.TOKEN_NEW)) then
    CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
    SYMBOL_TABLE.EXIT_SCOPE;
```

```
  SYMBOL_TABLE.UPDATE_SYM_TAB(0);
  if (P3.NAME) then
    if (P2.GENERIC_ACTUAL_PART) then
      null;
    end if;                                  -- if generic actual part
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Subprogram body");
    end if;                                   -- if match(token_semicolon)
  else
    P4.SYNTAX_ERROR("Subprogram body");
  end if;                                     -- if name statement
elsif (TM.MATCH(TM.TOKEN_SEPARATE)) then
  CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
  SYMBOL_TABLE.EXIT_SCOPE;
  SYMBOL_TABLE.UPDATE_SYM_TAB(0);
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    P4.SYNTAX_ERROR("Subprogram body");
  end if;                                     -- if match(token_semicolon)
elsif (TM.MATCH(TM.TOKEN_BRACKETS)) then
  CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
  SYMBOL_TABLE.EXIT_SCOPE;
  SYMBOL_TABLE.UPDATE_SYM_TAB(0);
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    P4.SYNTAX_ERROR("Subprogram body");
  end if;                                     -- if match(token_semicolon)
elsif (DECLARATIVE_PART) then
  LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  if (TM.MATCH(TM.TOKEN_BEGIN)) then
    TM.MATCHED_TOKEN(START_TOKEN);
    CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "BEGIN SUBPROGRAM");
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
    if (SEQUENCE_OF_STATEMENTS) then
      if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT = 0) then
        LOCATION_ONE := 0;
        CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
      else
        TM.MATCHED_TOKEN(STOP_TOKEN);
        LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
      end if;
      if (TM.MATCH(TM.TOKEN_EXCEPTION)) then
        if (EXCEPTION_HANDLER) then
          while (EXCEPTION_HANDLER) loop
```

```
          null;
        end loop;
      else
        P4.SYNTAX_ERROR("Subprogram body");
      end if;                                -- if exception_handler statement
    end if;                                  -- if match(token_exception)
  else
    P4.SYNTAX_ERROR("Subprogram body");
  end if;                                    -- if sequence of statements
end if;                                      -- if token begin
if (TM.MATCH(TM.TOKEN_END)) then
  TM.MATCHED_TOKEN(STOP_TOKEN);
  CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "END SUBPROGRAM");
  CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
  LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  if (SYMBOL_TABLE.FIND_LOCAL_KEY("END") = null) then
    raise SYMBOL_TABLE.REFERENCE_ERROR;
  else
    SYMBOL_TABLE.UPDATE_SYM_TAB(LOCATION_TWO);
  end if;
  if (LOCATION_ONE = 0) then
    NET_GENERATOR.EXPLICIT_END(LOCATION_TWO);
  else
    NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
  end if;
  CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
  if (P2.DESIGNATOR) then
    null;
  end if;                              -- if designator statement
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
    SYMBOL_TABLE.EXIT_SCOPE;
    return (TRUE);
  else
    P4.SYNTAX_ERROR("Subprogram body");
  end if;                              -- if match(token_semicolon)
else
  P4.SYNTAX_ERROR("Subprogram body");
end if;                                -- if match(token_end)
elsif (TM.MATCH(TM.TOKEN_BEGIN)) then
  TM.MATCHED_TOKEN(START_TOKEN);
  LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "BEGIN SUBPROGRAM");
  LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
  if (SEQUENCE_OF_STATEMENTS) then
    if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT = 0) then
      LOCATION_ONE := 0;
      CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
    else
      TM.MATCHED_TOKEN(STOP_TOKEN);
```

```
      LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
      CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
    end if;
    if (TM.MATCH(TM.TOKEN_EXCEPTION)) then
      if (EXCEPTION_HANDLER) then
        while (EXCEPTION_HANDLER) loop
          null;
        end loop;
      else
        P4.SYNTAX_ERROR("Subprogram body");
      end if;                                -- if exception_handler statement
    end if;                                  -- if match(token_exception)
  else
    P4.SYNTAX_ERROR("Subprogram body");
  end if;                                    -- if sequence of statements
  if (TM.MATCH(TM.TOKEN_END)) then
    TM.MATCHED_TOKEN(STOP_TOKEN);
    CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "END SUBPROGRAM");
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    if (SYMBOL_TABLE.FIND_LOCAL_KEY("END") = null) then
      raise SYMBOL_TABLE.REFERENCE_ERROR;
    else
      SYMBOL_TABLE.UPDATE_SYM_TAB(LOCATION_TWO);
    end if;
    if (LOCATION_ONE = 0) then
      NET_GENERATOR.EXPLICIT_END(LOCATION_TWO);
    else
      NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
    end if;
    CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
    if (P2.DESIGNATOR) then
      null;
    end if;                                -- if designator statement
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
      SYMBOL_TABLE.EXIT_SCOPE;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Subprogram body");
    end if;                                -- if match(token_semicolon)
  else
    P4.SYNTAX_ERROR("Subprogram body");
  end if;                                  -- if match(token_end)
elsif (TM.MATCH(TM.TOKEN_END)) then
  TM.MATCHED_TOKEN(STOP_TOKEN);
  CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "END SUBPROGRAM");
  CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
  LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  if (SYMBOL_TABLE.FIND_LOCAL_KEY("END") = null) then
    raise SYMBOL_TABLE.REFERENCE_ERROR;
```

```
    else
      SYMBOL_TABLE.UPDATE_SYM_TAB(LOCATION_TWO);
    end if;
    NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
    CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
    if (P2.DESIGNATOR) then
      null;
    end if;                                 -- if designator statement
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
      SYMBOL_TABLE.EXIT_SCOPE;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Subprogram body");
    end if;                                 -- if match(token_semicolon)
  elsif (P3.NAME) then
    CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
    SYMBOL_TABLE.EXIT_SCOPE;
    SYMBOL_TABLE.UPDATE_SYM_TAB(0);
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Subprogram body");
    end if;                                   -- if match(token_semicolon)
  else
    return (FALSE);
  end if;                                     -- if match(token_new)
end SUBPROGRAM_BODY;

--------------------------------------------------------------------------------

  -- FUNCTION_UNIT -->  DESIGNATOR [FORMAL_PART ?] return NAME is
  --                                                      SUBPROGRAM_BODY
  --                -->  DESIGNATOR [FORMAL_PART ?] return NAME ;
  --                -->  DESIGNATOR [FORMAL_PART ?] return NAME renames NAME ;
  --                -->  DESIGNATOR is SUBPROGRAM_BODY
  --                     (for generic instantiation)
function FUNCTION_UNIT return boolean is
START_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION : natural;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("FUNCTION_UNIT");
  end if;
  if (P2.DESIGNATOR) then
    TM.MATCHED_TOKEN(START_TOKEN);
    CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "FUNCTION CODE BLOCK");
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    LOCATION := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.LEXEME(1..START_TOKEN.LEXEME_SIZE),
                                SYMBOL_TABLE.FUNCTION_DECLARATION_TAG,
```

```
                            LOCATION);
SYMBOL_TABLE.INSERT_SYM_TAB("END", SYMBOL_TABLE.LABEL_NAME, 0);
if (IS_MAIN_PROGRAM) then
  NET_GENERATOR.START(SYMBOL_TABLE.FIND_KEY(START_TOKEN.LEXEME(1..
                                                   START_TOKEN.LEXEME_SIZE)));
  IS_MAIN_PROGRAM := FALSE;
end if;
if (P2.FORMAL_PART) then
  if (TM.MATCH(TM.TOKEN_RETURN)) then
    if (P3.NAME) then
      if (TM.MATCH(TM.TOKEN_IS)) then
        if (SUBPROGRAM_BODY) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Function unit");
        end if;
      elsif (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
        SYMBOL_TABLE.EXIT_SCOPE;
        SYMBOL_TABLE.UPDATE_SYM_TAB(0);
        return (TRUE);
      elsif (TM.MATCH(TM.TOKEN_RENAMES)) then
        CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
        SYMBOL_TABLE.EXIT_SCOPE;
        SYMBOL_TABLE.UPDATE_SYM_TAB(0);
        if (P3.NAME) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Function unit");
        end if;
      else
        P4.SYNTAX_ERROR("Function unit");
      end if;
    else
      P4.SYNTAX_ERROR("Function unit");
    end if;
  else
    P4.SYNTAX_ERROR("Function unit");
  end if;
elsif (TM.MATCH(TM.TOKEN_RETURN)) then
  if (P3.NAME) then
    if (TM.MATCH(TM.TOKEN_IS)) then
      if (SUBPROGRAM_BODY) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Function unit");
      end if;
    elsif (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
      SYMBOL_TABLE.INSERT_SYM_TAB("END", SYMBOL_TABLE.LABEL_NAME, 0);
      SYMBOL_TABLE.EXIT_SCOPE;
```

```
          SYMBOL_TABLE.UPDATE_SYM_TAB(0);
          return (TRUE);
        elsif (TM.MATCH(TM.TOKEN_RENAMES)) then
          CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
          SYMBOL_TABLE.EXIT_SCOPE;
          SYMBOL_TABLE.UPDATE_SYM_TAB(0);
          if (P3.NAME) then
            return (TRUE);
          else
            P4.SYNTAX_ERROR("Function unit");
          end if;
        else
          P4.SYNTAX_ERROR("Function unit");
        end if;
      else
        P4.SYNTAX_ERROR("Function unit");
      end if;
    else
      P4.SYNTAX_ERROR("Function unit");
    end if;
  elsif (TM.MATCH(TM.TOKEN_IS)) then
    if (SUBPROGRAM_BODY) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Function unit");
    end if;
  else
    return (FALSE);
  end if;
end FUNCTION_UNIT;

--------------------------------------------------------------------------------

  -- TASK_DECLARATION --> body identifier is SUBPROGRAM_BODY ;
  --                  --> [type ?] identifier [is [ENTRY_DECLARATION]*
  --                         [REPRESENTATION_CLAUSE]* end [identifier ?] ?] ;
function TASK_DECLARATION return boolean is
START_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION : natural;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("TASK_DECLARATION");
  end if;
  if (TM.MATCH(TM.TOKEN_TYPE)) then
    null;
  end if;                                 -- if match(token_type)
  if (TM.MATCH(TM.TOKEN_BODY)) then
    if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
      TM.MATCHED_TOKEN(START_TOKEN);
      CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "TASK CODE BLOCK");
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
```

```
    LOCATION := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.LEXEME(1..START_TOKEN.
                                LEXEME_SIZE), SYMBOL_TABLE.TASK_BODY_TAG,
                                LOCATION);
    if (TM.MATCH(TM.TOKEN_IS)) then
      if (SUBPROGRAM_BODY) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Package declaration");
      end if;                              -- if subprogram body
    else
      P4.SYNTAX_ERROR("Package declaration");
    end if;                                -- if token is
  else
    P4.SYNTAX_ERROR("Package declaration");
  end if;                                  -- if token identifier
elsif (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
  TM.MATCHED_TOKEN(START_TOKEN);
  SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.LEXEME(1..START_TOKEN.
                              LEXEME_SIZE),
                              SYMBOL_TABLE.TASK_DECLARATION_TAG, 0);
  SYMBOL_TABLE.INSERT_SYM_TAB("END", SYMBOL_TABLE.LABEL_NAME, 0);
  NET_GENERATOR.START(SYMBOL_TABLE.FIND_KEY(START_TOKEN.LEXEME(1..
                                            START_TOKEN.LEXEME_SIZE)));
  if (TM.MATCH(TM.TOKEN_IS)) then
    while (P2.ENTRY_DECLARATION) loop
      null;
    end loop;
    while (P2.REPRESENTATION_CLAUSE) loop
      null;
    end loop;
    if (TM.MATCH(TM.TOKEN_END)) then
      if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
        null;
      end if;                              -- if match(token_identifier)
      if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        SYMBOL_TABLE.EXIT_SCOPE;
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Task declaration");
      end if;                              -- if match(token_semicolon)
    else
      P4.SYNTAX_ERROR("Task declaration");
    end if;                                -- if match(token_end)
  elsif (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    SYMBOL_TABLE.EXIT_SCOPE;
    return (TRUE);
  else
    P4.SYNTAX_ERROR("Task declaration");
  end if;                                  -- if match(token_is)
else
```

```
    return (FALSE);
  end if;                                    -- if match(token_body)
end TASK_DECLARATION;

-------------------------------------------------------------------------------

  -- PACKAGE_DECLARATION -->  body identifier is SUBPROGRAM_BODY
  --                     -->  identifier is PACKAGE_TAIL_END
  --                     -->  identifier renames NAME;
function PACKAGE_DECLARATION return boolean is
START_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION : natural;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("PACKAGE_DECLARATION");
  end if;
  if (TM.MATCH(TM.TOKEN_BODY)) then
    if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
      TM.MATCHED_TOKEN(START_TOKEN);
      CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "PACKAGE CODE BLOCK");
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      LOCATION := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
      SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.LEXEME(1..START_TOKEN.
                              LEXEME_SIZE), SYMBOL_TABLE.PACKAGE_BODY_TAG,
                              LOCATION);
      if (TM.MATCH(TM.TOKEN_IS)) then
        if (SUBPROGRAM_BODY) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Package declaration");
        end if;                              -- if subprogram body
      else
        P4.SYNTAX_ERROR("Package declaration");
      end if;                                -- if token is
    else
      P4.SYNTAX_ERROR("Package declaration");
    end if;                                  -- if token identifier
  elsif (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    TM.MATCHED_TOKEN(START_TOKEN);
    if (TM.MATCH(TM.TOKEN_IS)) then
      SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.LEXEME(1..START_TOKEN.
                              LEXEME_SIZE),
                              SYMBOL_TABLE.PACKAGE_DECLARATION_TAG, 0);
      SYMBOL_TABLE.INSERT_SYM_TAB("END", SYMBOL_TABLE.LABEL_NAME, 0);
      if (PACKAGE_TAIL_END) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Package declaration");
      end if;                                -- if package_tail_end
    elsif (TM.MATCH(TM.TOKEN_RENAMES)) then
      if (P3.NAME) then
```

```
        if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Package declaration");
        end if;                               -- if token semicolon
      else
        P4.SYNTAX_ERROR("Package declaration");
      end if;                                 -- if name
    else
      P4.SYNTAX_ERROR("Package declaration");
    end if;                                   -- if token identifier
  else
    return (FALSE);
  end if;                                     -- if match(token_package)
end PACKAGE_DECLARATION;

-------------------------------------------------------------------------------

  -- PACKAGE_TAIL_END -->  new NAME [GENERIC_ACTUAL_PART ?] ;
  --                  -->  [BASIC_DECLARATIVE_ITEM]* [private
  --                         [BASIC_DECLARATIVE_ITEM]* ?] end [identifier ?] ;
function PACKAGE_TAIL_END return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("PACKAGE_TAIL_END");
  end if;
  if (TM.MATCH(TM.TOKEN_NEW)) then
    if (P3.NAME) then
      if (P2.GENERIC_ACTUAL_PART) then
        null;
      end if;                                 -- if generic_actual_part statement
      if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        SYMBOL_TABLE.EXIT_SCOPE;
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Package tail end");
      end if;                                 -- if match(token_semicolon)
    else
      P4.SYNTAX_ERROR("Package tail end");
    end if;                                   -- if name statement
  elsif (BASIC_DECLARATIVE_ITEM) then
    while (BASIC_DECLARATIVE_ITEM) loop
      null;
    end loop;
    if (TM.MATCH(TM.TOKEN_PRIVATE)) then
      while (BASIC_DECLARATIVE_ITEM) loop
        null;
      end loop;
    end if;                                   -- if match(token_private)
    if (TM.MATCH(TM.TOKEN_END)) then
      if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
```

```
        null;
      end if;
      if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        SYMBOL_TABLE.EXIT_SCOPE;
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Package tail end");
      end if;                               -- if match(token_semicolon)
    else
      P4.SYNTAX_ERROR("Package tail end");
    end if;                                 -- if match(token_end)
  elsif (TM.MATCH(TM.TOKEN_PRIVATE)) then
    while (BASIC_DECLARATIVE_ITEM) loop
      null;
    end loop;
    if (TM.MATCH(TM.TOKEN_END)) then
      if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
        null;
      end if;
      if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        SYMBOL_TABLE.EXIT_SCOPE;
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Package tail end");
      end if;                               -- if match(token_semicolon)
    else
      P4.SYNTAX_ERROR("Package tail end");
    end if;                                 -- if match(token_end)
  elsif (TM.MATCH(TM.TOKEN_END)) then
    if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
      null;
    end if;
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      SYMBOL_TABLE.EXIT_SCOPE;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Package tail end");
    end if;                                 -- if match(token_semicolon)
  else
    return (FALSE);
  end if;                                   -- if match(token_new)
end PACKAGE_TAIL_END;

------------------------------------------------------------------------------

  -- BASIC_DECLARATIVE_ITEM -->  BASIC_DECLARATIVE
  --                        -->  REPRESENTATION_CLAUSE
  --                        -->  use WITH_OR_USE_CLAUSE
function BASIC_DECLARATIVE_ITEM return boolean is
begin
  if (P4.PRINT_CALLS) then
```

```
    P4.OUT_PUT("BASIC_DECLARATIVE_ITEM");
  end if;
  if (BASIC_DECLARATION) then
    return (TRUE);
  elsif (P2.REPRESENTATION_CLAUSE) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_USE)) then
    if (P2.WITH_OR_USE_CLAUSE) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Basic declarative item");
    end if;
  else
    return (FALSE);
  end if;
end BASIC_DECLARATIVE_ITEM;

--------------------------------------------------------------------------------

  -- DECLARATIVE_PART-->  [BASIC_DECLARATIVE_ITEM]* [LATER_DECLARATIVE_ITEM]*
function DECLARATIVE_PART return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("DECLARATIVE_PART");
  end if;
  while (BASIC_DECLARATIVE_ITEM) loop
    null;
  end loop;
  while (LATER_DECLARATIVE_ITEM) loop
    null;
  end loop;
  return (TRUE);
end DECLARATIVE_PART;

--------------------------------------------------------------------------------

  -- BASIC_DECLARATION -->  type TYPE_DECLARATION
  --                   -->  subtype SUBTYPE_DECLARATION
  --                   -->  procedure PROCEDURE_UNIT
  --                   -->  function FUNCTION_UNIT
  --                   -->  package PACKAGE_DECLARATION
  --                   -->  generic GENERIC_DECLARATION
  --                   -->  IDENTIFIER_DECLARATION
  --                   -->  task TASK_DECLARATION
function BASIC_DECLARATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("BASIC_DECLARATION");
  end if;
  if (TM.MATCH(TM.TOKEN_TYPE)) then
    if (P2.TYPE_DECLARATION) then
```

```
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Basic declaration");
    end if;
  elsif (TM.MATCH(TM.TOKEN_SUBTYPE)) then
    if (P2.SUBTYPE_DECLARATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Basic declaration");
    end if;
  elsif (TM.MATCH(TM.TOKEN_PROCEDURE)) then
    if (PROCEDURE_UNIT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Basic declaration");
    end if;                               -- if procedure_unit statement
  elsif (TM.MATCH(TM.TOKEN_FUNCTION)) then
    if (FUNCTION_UNIT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Basic declaration");
    end if;                               -- if function_unit statement
  elsif (TM.MATCH(TM.TOKEN_PACKAGE)) then
    if (PACKAGE_DECLARATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Basic declaration");
    end if;                               -- if package_declaration
  elsif (TM.MATCH(TM.TOKEN_GENERIC)) then
    if (GENERIC_DECLARATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Basic declaration");
    end if;                               -- if generic_declaration
  elsif (P2.IDENTIFIER_DECLARATION) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_TASK)) then
    if (TASK_DECLARATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Basic declaration");
    end if;
  else
    return (FALSE);
  end if;
end BASIC_DECLARATION;

------------------------------------------------------------------------------

  -- LATER_DECLARATIVE_ITEM -->  PROPER_BODY
  --                        -->  generic GENERIC_DECLARATION
```

```
  --                          -->  use WITH_OR_USE_CLAUSE
function LATER_DECLARATIVE_ITEM return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("LATER_DECLARATIVE_ITEM");
  end if;
  if (PROPER_BODY) then                        -- check for body_declaration
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_GENERIC)) then
    if (GENERIC_DECLARATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Later declarative item");
    end if;                                    -- if generic_declaration
  elsif (TM.MATCH(TM.TOKEN_USE)) then
    if (P2.WITH_OR_USE_CLAUSE) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Later declarative item");
    end if;                                    -- if with_or_use_clause
  else
    return (FALSE);
  end if;
end LATER_DECLARATIVE_ITEM;

--------------------------------------------------------------------------------

  -- PROPER_BODY -->  procedure PROCEDURE_UNIT
  --             -->  function FUNCTION_UNIT
  --             -->  package PACKAGE_DECLARATION
  --             -->  task TASK_DECLARATION
function PROPER_BODY return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("PROPER_BODY");
  end if;
  if (TM.MATCH(TM.TOKEN_PROCEDURE)) then
    if (PROCEDURE_UNIT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Proper body");
    end if;                                    -- if procedure_unit statement
  elsif (TM.MATCH(TM.TOKEN_FUNCTION)) then
    if (FUNCTION_UNIT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Proper body");
    end if;                                    -- if function_unit statement
  elsif (TM.MATCH(TM.TOKEN_PACKAGE)) then
    if (PACKAGE_DECLARATION) then
      return (TRUE);
```

```
    else
      P4.SYNTAX_ERROR("Proper body");
    end if;                                    -- if package_declaration
  elsif (TM.MATCH(TM.TOKEN_TASK)) then
    if (TASK_DECLARATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Proper body");
    end if;
  else
    return (FALSE);
  end if;                                      -- if match(token_procedure)
end PROPER_BODY;

------------------------------------------------------------------------------

  -- SEQUENCE_OF_STATEMENTS -->  [STATEMENT]+
function SEQUENCE_OF_STATEMENTS return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SEQUENCE_OF_STATEMENTS");
  end if;
  if (STATEMENT) then
    while (STATEMENT) loop
      null;
    end loop;
    return (TRUE);
  else
    return (FALSE);
  end if;
end SEQUENCE_OF_STATEMENTS;

------------------------------------------------------------------------------

  -- STATEMENT -->  [LABEL ?] SIMPLE_STATEMENT
  --           -->  [LABEL ?] COMPOUND_STATEMENT
function STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("STATEMENT");
  end if;
  if (P2.LABEL) then
    null;
  end if;
  if (P2.SIMPLE_STATEMENT) then
    return (TRUE);
  elsif (COMPOUND_STATEMENT) then
    return (TRUE);
  else
    return (FALSE);
```

```
  end if;
end STATEMENT;

-----------------------------------------------------------------------------

  -- COMPOUND_STATEMENT -->  if IF_STATEMENT
  --                    -->  case CASE_STATEMENT
  --                    -->  LOOP_STATEMENT
  --                    -->  BLOCK_STATEMENT
  --                    -->  accept ACCEPT_STATEMENT
  --                    -->  SELECT_STATEMENT
function COMPOUND_STATEMENT return boolean is
START_TOKEN  : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE : positive;
LOCATION_TWO : positive;
use SYMBOL_TABLE;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("COMPOUND_STATEMENT");
  end if;
  if (TM.MATCH(TM.TOKEN_IF)) then
    if (IF_STATEMENT) then
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Compound statement");
    end if;                                -- if if_statement
  elsif (TM.MATCH(TM.TOKEN_CASE)) then
    if (CASE_STATEMENT) then
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Compound statement");
    end if;                                -- if case_statement
  elsif (LOOP_STATEMENT) then
    return (TRUE);
  elsif (BLOCK_STATEMENT) then
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    return (TRUE);
  elsif(TM.MATCH(TM.TOKEN_ACCEPT)) then
    if (ACCEPT_STATEMENT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Compound statement");
    end if;                                -- if accept_statement
  elsif (SELECT_STATEMENT) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end COMPOUND_STATEMENT;
```

```
--------------------------------------------------------------------------------

  -- BLOCK_STATEMENT -->   [declare DECLARATIVE_PART ?]
  --                        begin SEQUENCE_OF_STATEMENTS [exception
  --                        [EXCEPTION_HANDLER]+ ?] ?] end [identifier ?] ;
function BLOCK_STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("BLOCK_STATEMENT");
  end if;
  if (TM.MATCH(TM.TOKEN_DECLARE)) then
    if (DECLARATIVE_PART) then
      null;
    else
      P4.SYNTAX_ERROR("Block statement");
    end if;                               -- if declarative_part statement
  end if;                                 -- if match(token_declare)
  if (TM.MATCH(TM.TOKEN_BEGIN)) then
    if (SEQUENCE_OF_STATEMENTS) then
      if (TM.MATCH(TM.TOKEN_EXCEPTION)) then
        if (EXCEPTION_HANDLER) then
          while (EXCEPTION_HANDLER) loop
            null;
          end loop;
        else
          P4.SYNTAX_ERROR("Block statement");
        end if;                           -- if exception_handler statement
      end if;                             -- if match(token_exception)
      if (TM.MATCH(TM.TOKEN_END)) then
        if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
          null;
        end if;                           -- if match(token_identifier)
        if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Block statement");
        end if;                           -- if match(token_semicolon)
      else
        P4.SYNTAX_ERROR("Block statement");
      end if;                             -- if match(token_end)
    else
      P4.SYNTAX_ERROR("Block statement");
    end if;                               -- if sequence_of_statements
  else
    return (FALSE);
  end if;                                 -- if match(token_begin)
end BLOCK_STATEMENT;

--------------------------------------------------------------------------------
```

```
  -- IF_STATEMENT -->  EXPRESSION then SEQUENCE_OF_STATEMENTS
  --                     [elsif EXPRESSION then SEQUENCE_OF_STATEMENTS]*
  --                     [else SEQUENCE_OF_STATEMENTS ?] end if ;
function IF_STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("IF_STATEMENT");
  end if;
  if (P3.EXPRESSION) then
    if (TM.MATCH(TM.TOKEN_THEN)) then
      if (SEQUENCE_OF_STATEMENTS) then
        while (TM.MATCH(TM.TOKEN_ELSIF)) loop
          if (P3.EXPRESSION) then
            if (TM.MATCH(TM.TOKEN_THEN)) then
              if not (SEQUENCE_OF_STATEMENTS) then
                P4.SYNTAX_ERROR("If statement");
              end if;                       -- if not sequence_of_statements
            else
              P4.SYNTAX_ERROR("If statement");
            end if;                         -- if match(token_then)
          else
            P4.SYNTAX_ERROR("If statement");
          end if;                           -- if expression statement
        end loop;
        if (TM.MATCH(TM.TOKEN_ELSE)) then
          if (SEQUENCE_OF_STATEMENTS) then
            null;
          else
            P4.SYNTAX_ERROR("If statement");
          end if;                           -- if sequence_of_statements
        end if;                             -- if match(token_else)
        if (TM.MATCH(TM.TOKEN_END)) then
          if (TM.MATCH(TM.TOKEN_IF)) then
            if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
              return (TRUE);
            else
              P4.SYNTAX_ERROR("If statement");
            end if;                         -- if match(token_semicolon)
          else
            P4.SYNTAX_ERROR("If statement");
          end if;                           -- if match(token_if)
        else
          P4.SYNTAX_ERROR("If statement");
        end if;                             -- if match(token_end)
      else
        P4.SYNTAX_ERROR("If statement");
      end if;                               -- if sequence_of_statements
    else
      P4.SYNTAX_ERROR("If statement");
    end if;                                 -- if match(token_then)
  else
```

```
    return (FALSE);
  end if;                                    -- if expression statement
end IF_STATEMENT;

-------------------------------------------------------------------------------

 -- CASE_STATEMENT -->  EXPRESSION is [CASE_STATEMENT_ALTERNATIVE]+ end case ;
function CASE_STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("CASE_STATEMENT");
  end if;
  if (P3.EXPRESSION) then
    if (TM.MATCH(TM.TOKEN_IS)) then
      if (CASE_STATEMENT_ALTERNATIVE) then
        while (CASE_STATEMENT_ALTERNATIVE) loop
          null;
        end loop;
        if (TM.MATCH(TM.TOKEN_END)) then
          if (TM.MATCH(TM.TOKEN_CASE)) then
            if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
              return (TRUE);
            else
              P4.SYNTAX_ERROR("Case statement");
            end if;                          -- if match(token_semicolon)
          else
            P4.SYNTAX_ERROR("Case statement");
          end if;                            -- if match(token_case)
        else
          P4.SYNTAX_ERROR("Case statement");
        end if;                              -- if match(token_end)
      else
        P4.SYNTAX_ERROR("Case statement");
      end if;                                -- if case_statement_alternative
    else
      P4.SYNTAX_ERROR("Case statement");
    end if;                                  -- if match(token_is)
  else
    return (FALSE);
  end if;                                    -- if expression statement
end CASE_STATEMENT;

-------------------------------------------------------------------------------

  -- CASE_STATEMENT_ALTERNATIVE -->  when CHOICE [| CHOICE]* =>
  --                                   SEQUENCE_OF_STATEMENTS
function CASE_STATEMENT_ALTERNATIVE return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("CASE_STATEMENT_ALTERNATIVE");
  end if;
```

```
  if (TM.MATCH(TM.TOKEN_WHEN)) then
    if (P3.CHOICE) then
      while (TM.MATCH(TM.TOKEN_BAR)) loop
        if not (P3.CHOICE) then
          P4.SYNTAX_ERROR("Case statement alternative");
        end if;                              -- if not choice statement
      end loop;
      if (TM.MATCH(TM.TOKEN_ARROW)) then
        if (SEQUENCE_OF_STATEMENTS) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Case statement alternative");
        end if;                              -- if sequence_of_statements
      else
        P4.SYNTAX_ERROR("Case statement alternative");
      end if;                                -- if match(token_arrow)
    else
      P4.SYNTAX_ERROR("Case statement alternative");
    end if;                                  -- if choice statement
  else
    return (FALSE);
  end if;                                    -- if match(token_when)
end CASE_STATEMENT_ALTERNATIVE;

--------------------------------------------------------------------------------

  -- LOOP_STATEMENT -->    [ITERATION_SCHEME ?] loop
  --                       SEQUENCE_OF_STATEMENTS end loop [identifier ?] ;
function LOOP_STATEMENT return boolean is
STOP_TOKEN   : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE : natural;
LOCATION_TWO : positive;
use SYMBOL_TABLE;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("LOOP_STATEMENT");
  end if;
  if (P3.ITERATION_SCHEME) then
    null;
  end if;                                    -- if iteration_scheme statement
  if (TM.MATCH(TM.TOKEN_LOOP)) then
    TM.MATCHED_TOKEN(STOP_TOKEN);
    if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT /= 0) then
      LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
      CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
      CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "LOOP BLOCK");
      LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
      SYMBOL_TABLE.INSERT_SYM_TAB("LOOP", LOOP_TAG, LOCATION_TWO);
      SYMBOL_TABLE.INSERT_SYM_TAB("END", LABEL_NAME, 0);
```

```
else
  CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
  CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "LOOP BLOCK");
  CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
  LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  SYMBOL_TABLE.INSERT_SYM_TAB("LOOP", LOOP_TAG, LOCATION_TWO);
  SYMBOL_TABLE.INSERT_SYM_TAB("END", LABEL_NAME, 0);
end if;
if (SEQUENCE_OF_STATEMENTS) then
  if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT = 0) then
    LOCATION_ONE := 0;
    CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
  else
    TM.MATCHED_TOKEN(STOP_TOKEN);
    LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
  end if;
  if (TM.MATCH(TM.TOKEN_END)) then
    if (TM.MATCH(TM.TOKEN_LOOP)) then
      TM.MATCHED_TOKEN(STOP_TOKEN);
      CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "END LOOP");
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
      if (SYMBOL_TABLE.FIND_LOCAL_KEY("END") = null) then
        raise SYMBOL_TABLE.REFERENCE_ERROR;
      else
        SYMBOL_TABLE.UPDATE_SYM_TAB(LOCATION_TWO);
      end if;
      if (LOCATION_ONE = 0) then
        NET_GENERATOR.EXPLICIT_END(LOCATION_TWO);
      else
        NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
      end if;
      CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
      CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "");
      if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
        null;
      end if;                           -- if match(token_identifier)
      if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        SYMBOL_TABLE.EXIT_SCOPE;
        NET_GENERATOR.END_LOOP(LOCATION_TWO, SYMBOL_TABLE.RETRIEVE_SYM);
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Loop statement: expecting semicolon");
      end if;                           -- if match(token_semicolon)
    else
      P4.SYNTAX_ERROR("Loop statement: end must be fully bracketed");
    end if;                             -- if match(token_loop)
  else
    P4.SYNTAX_ERROR("Loop statement: expecting 'end'");
  end if;                               -- if match(token_end)
```

```
    else
      P4.SYNTAX_ERROR("Loop statement: expecting sequence of statements");
    end if;                                 -- if sequence_of_statements
  else
    return (FALSE);
  end if;                                   -- if match(token_loop)
end LOOP_STATEMENT;

--------------------------------------------------------------------------------

  -- EXCEPTION_HANDLER -->  when EXCEPTION_CHOICE [| EXCEPTION_CHOICE]* =>
  --                          SEQUENCE_OF_STATEMENTS
function EXCEPTION_HANDLER return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("EXCEPTION_HANDLER");
  end if;
  if (TM.MATCH(TM.TOKEN_WHEN)) then
    if (P2.EXCEPTION_CHOICE) then
      while (TM.MATCH(TM.TOKEN_BAR)) loop
        if not (P2.EXCEPTION_CHOICE) then
          P4.SYNTAX_ERROR("Exception handler");
        end if;                             -- if not exception_choice
      end loop;
      if (TM.MATCH(TM.TOKEN_ARROW)) then
        if (SEQUENCE_OF_STATEMENTS) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Exception handler");
        end if;                             -- if sequence_of_statements
      else
        P4.SYNTAX_ERROR("Exception handler");
      end if;                               -- if match(token_arrow)
    else
      P4.SYNTAX_ERROR("Exception handler");
    end if;                                 -- if exception_choice statement
  else
    return (FALSE);
  end if;                                   -- if match(token-when)
end EXCEPTION_HANDLER;

--------------------------------------------------------------------------------

  -- ACCEPT_STATEMENT -->  identifier [(EXPRESSION) ?] [FORMAL_PART ?]
  --                        [do SEQUENCE_OF_STATEMENTS end [identifier ?] ?] ;
function ACCEPT_STATEMENT return boolean is
STOP_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE : natural;
LOCATION_TWO : positive;
use SYMBOL_TABLE;
begin
```

```
if (P4.PRINT_CALLS) then
  P4.OUT_PUT("ACCEPT_STATEMENT");
end if;
if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
  TM.MATCHED_TOKEN(STOP_TOKEN);
  if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT /= 0) then
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  else
    CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
    CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "ACCEPT STATEMENT");
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  end if;
  CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "ENTRY BLOCK");
  LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
  NET_GENERATOR.TASK_ACCEPT(LOCATION_ONE, LOCATION_TWO);
  SYMBOL_TABLE.INSERT_SYM_TAB(STOP_TOKEN.LEXEME(1..STOP_TOKEN.
                              LEXEME_SIZE), SYMBOL_TABLE.ACCEPT_TAG,
                              LOCATION_TWO);
  CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
  if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    if (P3.EXPRESSION) then
      if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
        null;
      else
        P4.SYNTAX_ERROR("Accept statement");
      end if;                            -- if match(token_right_paren)
    else
      P4.SYNTAX_ERROR("Accept statement");
    end if;                              -- if expression statement
  end if;                                -- if match(token_left_paren)
  if (P2.FORMAL_PART) then
    null;
  end if;                                -- if formal_part statement
  if (TM.MATCH(TM.TOKEN_DO)) then
    TM.MATCHED_TOKEN(STOP_TOKEN);
    CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
    CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE,
                                  "BEGIN ACCEPT STATEMENTS");
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    if (SEQUENCE_OF_STATEMENTS) then
      if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT = 0) then
        LOCATION_ONE := 0;
        CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
      else
        TM.MATCHED_TOKEN(STOP_TOKEN);
        LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
      end if;
```

```
      if (TM.MATCH(TM.TOKEN_END)) then
        TM.MATCHED_TOKEN(STOP_TOKEN);
        CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "END ENTRY BLOCK");
        CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
        LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        if (SYMBOL_TABLE.FIND_LOCAL_KEY("END") = null) then
          raise SYMBOL_TABLE.REFERENCE_ERROR;
        else
          SYMBOL_TABLE.UPDATE_SYM_TAB(LOCATION_TWO);
        end if;
        if (LOCATION_ONE = 0) then
          NET_GENERATOR.EXPLICIT_END_ACCEPT(LOCATION_TWO);
        else
          NET_GENERATOR.END_ACCEPT(LOCATION_ONE, LOCATION_TWO);
        end if;
        CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
        CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "");
        if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
          null;
        end if;                           -- if match(token_identifier)
      else
        P4.SYNTAX_ERROR("Accept statement");
      end if;                             -- if match(token_end)
    else
      P4.SYNTAX_ERROR("Accept statement");
    end if;                               -- if sequence_of_statements
  end if;                                 -- if match(token_do)
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    SYMBOL_TABLE.EXIT_SCOPE;
    return (TRUE);
  else
    P4.SYNTAX_ERROR("Accept statement");
  end if;                                 -- if match(token_semicolon)
 else
  return (FALSE);
 end if;                                  -- if match(token_identifier)
end ACCEPT_STATEMENT;

---------------------------------------------------------------------------------

 -- SELECT_STATEMENT -->  select SELECT_STATEMENT_TAIL [ SELECT_ENTRY_CALL ?]
 --                         end select ;
function SELECT_STATEMENT return boolean is
STOP_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE : positive;
LOCATION_TWO : positive;
use SYMBOL_TABLE;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SELECT_STATEMENT");
  end if;
```

```
if (TM.MATCH(TM.TOKEN_SELECT)) then
  TM.MATCHED_TOKEN(STOP_TOKEN);
  if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT /= 0) then
    LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
    CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "SELECT BLOCK");
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
  else
    CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
    CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "SELECT BLOCK");
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  end if;
  SYMBOL_TABLE.INSERT_SYM_TAB("SELECT", SELECT_TAG, LOCATION_TWO);
  SYMBOL_TABLE.INSERT_SYM_TAB("END", LABEL_NAME, 0);
  NET_GENERATOR.DECISION_START(LOCATION_TWO, SYMBOL_TABLE.RETRIEVE_SYM);
  if (SELECT_STATEMENT_TAIL) then
    if (SELECT_ENTRY_CALL) then
      if (TM.MATCH(TM.TOKEN_END)) then
        if (TM.MATCH(TM.TOKEN_SELECT)) then
          if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
            TM.MATCHED_TOKEN(STOP_TOKEN);
            if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT /= 0) then
              LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
              CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
              NET_GENERATOR.END_DECISION(LOCATION_ONE);
            else
              CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
              NET_GENERATOR.EXPLICIT_END_DECISION;
            end if;
            CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "END SELECT");
            CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
            LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
            if (SYMBOL_TABLE.FIND_LOCAL_KEY("END") = null) then
              raise SYMBOL_TABLE.REFERENCE_ERROR;
            else
              SYMBOL_TABLE.UPDATE_SYM_TAB(LOCATION_ONE);
            end if;
            CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
            CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "");
            SYMBOL_TABLE.EXIT_SCOPE;
            return (TRUE);
          else
            P4.SYNTAX_ERROR("Select statement");
          end if;                             -- if match(token_semicolon)
        else
          P4.SYNTAX_ERROR("Select statement");
        end if;                               -- if match(token_select)
      else
```

```
        P4.SYNTAX_ERROR("Select statement");
      end if;                                 -- if match(token_end)
    elsif (TM.MATCH(TM.TOKEN_END)) then
      if (TM.MATCH(TM.TOKEN_SELECT)) then
        if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
          TM.MATCHED_TOKEN(STOP_TOKEN);
          if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT /= 0) then
            LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
            CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
            NET_GENERATOR.END_DECISION(LOCATION_ONE);
          else
            CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
            NET_GENERATOR.EXPLICIT_END_DECISION;
          end if;
          CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "END SELECT");
          CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
          LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
          if (SYMBOL_TABLE.FIND_LOCAL_KEY("END") = null) then
            raise SYMBOL_TABLE.REFERENCE_ERROR;
          else
            SYMBOL_TABLE.UPDATE_SYM_TAB(LOCATION_ONE);
          end if;
          CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
          CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "");
          SYMBOL_TABLE.EXIT_SCOPE;
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Select statement");
        end if;                               -- if match(token_semicolon)
      else
        P4.SYNTAX_ERROR("Select statement");
      end if;                                 -- if match(token_select)
    else
      P4.SYNTAX_ERROR("Select statement");
    end if;                                   -- if match(token_end)
  else
    P4.SYNTAX_ERROR("Select statement");
  end if;                                     -- if select_statement_tail
else
  return (FALSE);
end if;
end SELECT_STATEMENT;

--------------------------------------------------------------------------------

  -- SELECT_STATEMENT_TAIL -->  SELECT_ALTERNATIVE [or SELECT_ALTERNATIVE]*
  --                       -->  NAME ; [SEQUENCE_OF_STATEMENTS ?]
function SELECT_STATEMENT_TAIL return boolean is
STOP_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE : positive;
SEARCH_POINTER : SYMBOL_TABLE.SYM_TAB_ACCESS;
```

```
use SYMBOL_TABLE;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SELECT_STATEMENT_TAIL");
  end if;
  if (SELECT_ALTERNATIVE) then
    while (TM.MATCH(TM.TOKEN_OR)) loop
      TM.MATCHED_TOKEN(STOP_TOKEN);
      if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT /= 0) then
        LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
        NET_GENERATOR.DECISION_OR(LOCATION_ONE);
      else
        CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
        NET_GENERATOR.EXPLICIT_DECISION_OR;
      end if;
      if not (SELECT_ALTERNATIVE) then
        P4.SYNTAX_ERROR("Select statement tail");
      end if;
    end loop;
    return (TRUE);
  else
    SYMBOL_TABLE.SAVE_CURRENT_ENTRY;
    if (P3.NAME) then                        -- check for entry call statement
      TM.MATCHED_TOKEN(STOP_TOKEN);
      SEARCH_POINTER := SYMBOL_TABLE.RETRIEVE_SYM;
      if ((SEARCH_POINTER /= null) and then
      (SEARCH_POINTER.TAG_TYPE = SYMBOL_TABLE.ENTRY_TAG)) then
        LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
        CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
        NET_GENERATOR.ENTRY_CALL(LOCATION_ONE, SEARCH_POINTER);
        CODE_BLOCKER.ENTER_CODE_BLOCK(STOP_TOKEN.SOURCE, "");
        SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
      else
        SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
        return (FALSE);
      end if;
      if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        if (SEQUENCE_OF_STATEMENTS) then
          null;
        end if;                              -- if sequence_of_statements
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Select statement tail");
      end if;                                -- if match(token_semicolon)
    else
      return (FALSE);
    end if;
  end if;                                    -- if select_alternative statement
end SELECT_STATEMENT_TAIL;
```

```
--------------------------------------------------------------------------------

  -- SELECT_ALTERNATIVE -->  [when EXPRESSION => ?] accept ACCEPT_STATEMENT
  --                           [SEQUENCE_OF_STATEMENTS ?]
  --                    -->  [when EXPRESSION => ?] delay DELAY_STATEMENT
  --                           [SEQUENCE_OF_STATEMENTS ?]
  --                    -->  [when EXPRESSION => ?] terminate ;
function SELECT_ALTERNATIVE return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SELECT_ALTERNATIVE");
  end if;
  if (TM.MATCH(TM.TOKEN_WHEN)) then
    if (P3.EXPRESSION) then
      if (TM.MATCH(TM.TOKEN_ARROW)) then
        null;
      else
        P4.SYNTAX_ERROR("Select alternative");
      end if;                               -- if match(token_arrow)
    else
      P4.SYNTAX_ERROR("Select alternative");
    end if;                                 -- if expression statement
  end if;                                   -- if match(token_when)
  if (TM.MATCH(TM.TOKEN_ACCEPT)) then
    if (ACCEPT_STATEMENT) then
      if (SEQUENCE_OF_STATEMENTS) then
        null;
      end if;                               -- if sequence_of_statements
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Select alternative");
    end if;                                 -- if accept_statement
  elsif (TM.MATCH(TM.TOKEN_DELAY)) then
    if (P3.DELAY_STATEMENT) then
      if (SEQUENCE_OF_STATEMENTS) then
        null;
      end if;                               -- if sequence_of_statements
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Select alternative");
    end if;                                 -- if delay_statement
  elsif (TM.MATCH(TM.TOKEN_TERMINATE)) then
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Select alternative");
    end if;                                 -- if match(token_semicolon)
  else
    return (FALSE);
```

```
  end if;                                    -- if match(token_accept)
end SELECT_ALTERNATIVE;

--------------------------------------------------------------------------------

  -- SELECT_ENTRY_CALL -->  else SEQUENCE_OF_STATEMENTS
  --                   -->  or delay DELAY_STATEMENT [SEQUENCE_OF_STATEMENTS ?]
function SELECT_ENTRY_CALL return boolean is
STOP_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE : positive;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SELECT_ENTRY_CALL");
  end if;
  if (TM.MATCH(TM.TOKEN_ELSE)) then
    TM.MATCHED_TOKEN(STOP_TOKEN);
    if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT /= 0) then
      LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
      CODE_BLOCKER.EXIT_CODE_BLOCK(STOP_TOKEN.SOURCE);
      NET_GENERATOR.DECISION_OR(LOCATION_ONE);
    else
      CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
      NET_GENERATOR.EXPLICIT_DECISION_OR;
    end if;
    if (SEQUENCE_OF_STATEMENTS) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Select entry call");
    end if;                                  -- if sequence_of_statements
  elsif (TM.MATCH(TM.TOKEN_OR)) then
    if (TM.MATCH(TM.TOKEN_DELAY)) then
      if (P3.DELAY_STATEMENT) then
        if (SEQUENCE_OF_STATEMENTS) then
          null;
        end if;                              -- if sequence_of_statements
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Select entry call");
      end if;                                -- if delay_statement
    else
      P4.SYNTAX_ERROR("Select entry call");
    end if;                                  -- if match(token_delay)
  else
    return (FALSE);
  end if;                                    -- if match(token_else)
end SELECT_ENTRY_CALL;

end PARSER_1;
```

```
--***********************************************************--
--                                                           --
--  TITLE:          ADAFLOW                                  --
--                                                           --
--  MODULE NAME:    PACKAGE PARSER_2                         --
--  FILE NAME:      PARSER2.ADS                              --
--                                                           --
--  DATE CREATED:   20 FEB 88                                --
--  LAST MODIFIED:  28 APR 88                                --
--                                                           --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                 --
--                                                           --
--  BASED ON A MODIFIED ADA GRAMMAR DEVELOPED BY:            --
--                  LCDR JEFFREY L. NIEDER, USN              --
--                  LT KARL S. FAIRBANKS, JR., USN           --
--                  LCDR PAUL M. HERZIG, USN                 --
--                                                           --
--  DESCRIPTION:  This package defines the functions         --
--       that are the middle level productions for a top-down, --
--       recursive descent parser.                           --
--                                                           --
--***********************************************************--

package PARSER_2 is
  function GENERIC_ACTUAL_PART return boolean;
  function GENERIC_ASSOCIATION return boolean;
  function GENERIC_FORMAL_PARAMETER return boolean;
  function GENERIC_TYPE_DEFINITION return boolean;
  function PRIVATE_TYPE_DECLARATION return boolean;
  function TYPE_DECLARATION return boolean;
  function SUBTYPE_DECLARATION return boolean;
  function DISCRIMINANT_PART return boolean;
  function DISCRIMINANT_SPECIFICATION return boolean;
  function TYPE_DEFINITION return boolean;
  function RECORD_TYPE_DEFINITION return boolean;
  function COMPONENT_LIST return boolean;
  function COMPONENT_DECLARATION return boolean;
  function VARIANT_PART return boolean;
  function VARIANT return boolean;
  function WITH_OR_USE_CLAUSE return boolean;
  function FORMAL_PART return boolean;
  function IDENTIFIER_DECLARATION return boolean;
  function IDENTIFIER_DECLARATION_TAIL return boolean;
  function EXCEPTION_TAIL return boolean;
  function EXCEPTION_CHOICE return boolean;
  function CONSTANT_TERM return boolean;
  function IDENTIFIER_TAIL return boolean;
  function PARAMETER_SPECIFICATION return boolean;
  function IDENTIFIER_LIST return boolean;
  function MODE return boolean;
  function DESIGNATOR return boolean;
```

```
  function SIMPLE_STATEMENT return boolean;
  function ASSIGNMENT_OR_PROCEDURE_CALL return boolean;
  function LABEL return boolean;
  function ENTRY_DECLARATION return boolean;
  function REPRESENTATION_CLAUSE return boolean;
  function RECORD_REPRESENTATION_CLAUSE return boolean;
end PARSER_2;
```

```
--*************************************************************--
--                                                             --
--  TITLE:          ADAFLOW                                    --
--                                                             --
--  MODULE NAME:    PACKAGE PARSER_2                           --
--  FILE NAME:      PARSER2.ADB                                --
--                                                             --
--  DATE CREATED:   20 FEB 88                                  --
--  LAST MODIFIED:  28 APR 88                                  --
--                                                             --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                   --
--                                                             --
--  BASED ON A MODIFIED ADA GRAMMAR DEVELOPED BY:              --
--                  LCDR JEFFREY L. NIEDER, USN                --
--                  LT KARL S. FAIRBANKS, JR., USN             --
--                  LCDR PAUL M. HERZIG, USN                   --
--                                                             --
--  DESCRIPTION:  This package implements the functions        --
--       that are the middle level productions for a top-down, --
--       recursive descent parser.  Each function is preceded  --
--       by the grammar productions they are implementing.     --
--                                                             --
--*************************************************************--

with PARSER_3, PARSER_4, TOKEN_MATCHER, TOKEN_SCANNER,
     CODE_BLOCKER, SYMBOL_TABLE, NET_GENERATOR;

package body PARSER_2 is
  package TM renames TOKEN_MATCHER;
  package P3 renames PARSER_3;
  package P4 renames PARSER_4;

  -- GENERIC_ACTUAL_PART -->  (GENERIC_ASSOCIATION [, GENERIC_ASSOCIATION]* )
function GENERIC_ACTUAL_PART return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("GENERIC_ACTUAL_PART");
  end if;
  if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    if (GENERIC_ASSOCIATION) then
      while (TM.MATCH(TM.TOKEN_COMMA)) loop
        if not (GENERIC_ASSOCIATION) then
          P4.SYNTAX_ERROR("Generic actual part");
        end if;                          -- if not generic_association
      end loop;
      if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Generic actual part");
      end if;                            -- if match(token_right_paren)
    else
```

```
      P4.SYNTAX_ERROR("Generic actual part");
    end if;                                   -- if generic association statement
  else
    return(FALSE);
  end if;                                     -- if match(token_left_paren)
end GENERIC_ACTUAL_PART;

--------------------------------------------------------------------------------

  -- GENERIC_ASSOCIATION -->  [GENERIC_FORMAL_PARAMETER ?] EXPRESSION
function GENERIC_ASSOCIATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("GENERIC_ASSOCIATION");
  end if;
  if (GENERIC_FORMAL_PARAMETER) then
    null;
  end if;                                     -- if generic_formal_parameter
  if (P3.EXPRESSION) then                     -- check generic_actual_parameter
    return (TRUE);
  else
    return (FALSE);
  end if;                                     -- if expression
end GENERIC_ASSOCIATION;

--------------------------------------------------------------------------------

  -- GENERIC_FORMAL_PARAMETER -->  identifier =>
  --                          -->  string_literal =>
function GENERIC_FORMAL_PARAMETER return boolean is
PEEK_AHEAD_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
TEST_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
use TOKEN_SCANNER;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("GENERIC_FORMAL_PARAMETER");
  end if;
  TEST_TOKEN.LEXEME := (others => ' ');
  TEST_TOKEN.LEXEME(1..2) := "=>";
  TEST_TOKEN.LEXEME_SIZE := 2;
  TEST_TOKEN.TOKEN_TYPE := TOKEN_SCANNER.DELIMITER;
  TM.NEXT_TOKEN(PEEK_AHEAD_TOKEN);
  if (PEEK_AHEAD_TOKEN = TEST_TOKEN) then
    if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
      if (TM.MATCH(TM.TOKEN_ARROW)) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Generic formal parameter");
      end if;                                 -- if match(token_arrow)
    elsif (TM.MATCH(TM.TOKEN_STRING_LITERAL)) then
      if (TM.MATCH(TM.TOKEN_ARROW)) then
```

```
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Generic formal parameter");
      end if;                                        -- if match(token_arrow)
    else
      P4.SYNTAX_ERROR("Generic formal parameter");
    end if;                                          -- if match(token_identifier)
  else
    return (FALSE);
  end if;                                            -- if lookahead_token = "=>"
end GENERIC_FORMAL_PARAMETER;

--------------------------------------------------------------------------------

  -- GENERIC_TYPE_DEFINITION -->  ( <> )
  --                         -->  range <>
  --                         -->  digits <>
  --                         -->  delta <>
  --                         -->  array ARRAY_TYPE_DEFINITION
  --                         -->  access SUBTYPE_INDICATION
function GENERIC_TYPE_DEFINITION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("GENERIC_TYPE_DEFINITION");
  end if;
  if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    if (TM.MATCH(TM.TOKEN_BRACKETS)) then
      if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Generic type definition");
      end if;                                        -- if match(token_right_paren)
    else
      P4.SYNTAX_ERROR("Generic type definition");
    end if;                                          -- if match(token_brackets)
  elsif (TM.MATCH(TM.TOKEN_RANGE)) or else (TM.MATCH(TM.TOKEN_DIGITS))
    or else (TM.MATCH(TM.TOKEN_DELTA)) then
    if (TM.MATCH(TM.TOKEN_BRACKETS)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Generic type definition");
    end if;                                          -- if match(token_brackets)
  elsif (TM.MATCH(TM.TOKEN_ARRAY)) then
    if (P3.ARRAY_TYPE_DEFINITION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Generic type definition");
    end if;                                          -- if array_type_definition
  elsif (TM.MATCH(TM.TOKEN_ACCESS)) then
    if (P3.SUBTYPE_INDICATION) then
      return (TRUE);
```

```
    else
      P4.SYNTAX_ERROR("Generic type definition");
    end if;                                   -- if subtype_indication
  else
    return (FALSE);
  end if;                                     -- if match(token_left_paren)
end GENERIC_TYPE_DEFINITION;

-------------------------------------------------------------------------------

  -- PRIVATE_TYPE_DECLARATION -->  [limited ?] private
function PRIVATE_TYPE_DECLARATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("PRIVATE_TYPE_DECLARATION");
  end if;
  if (TM.MATCH(TM.TOKEN_LIMITED)) then
    null;
  end if;
  if (TM.MATCH(TM.TOKEN_PRIVATE)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end PRIVATE_TYPE_DECLARATION;

-------------------------------------------------------------------------------

  -- SUBTYPE_DECLARATION -->  identifier is SUBTYPE_INDICATION ;
function SUBTYPE_DECLARATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SUBTYPE_DECLARATION");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    if (TM.MATCH(TM.TOKEN_IS)) then
      if (P3.SUBTYPE_INDICATION) then
        if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Subtype declaration");
        end if;                               -- if match(token_semicolon)
      else
        P4.SYNTAX_ERROR("Subtype declaration");
      end if;                                 -- if subtype_indication
    else
      P4.SYNTAX_ERROR("Subtype declaration");
    end if;                                   -- if match(token_is)
  else
    return (FALSE);
```

```
  end if;                                       -- if match(token_identifier)
end SUBTYPE_DECLARATION;

--------------------------------------------------------------------------------

  -- TYPE_DECLARATION -->  identifier [DISCRIMINANT_PART ?]
  --                         [is PRIVATE_TYPE_DECLARATION ?];
  --                   -->  identifier [DISCRIMINANT_PART ?]
  --                         [is TYPE_DEFINITION ?];
function TYPE_DECLARATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("TYPE_DECLARATION");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    if (DISCRIMINANT_PART) then
      null;
    end if;                                     -- if discriminant_part
    if (TM.MATCH(TM.TOKEN_IS)) then             -- declaration is full_type if 'is'
      if (PRIVATE_TYPE_DECLARATION) then
        null;
      elsif (TYPE_DEFINITION) then              -- present else incomplete_type
        null;
      else
        P4.SYNTAX_ERROR("Type declaration");
      end if;                                   -- if type_definition
    end if;                                     -- if match(token_is)
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Type declaration");
    end if;                                     -- if match(token_semicolon)
  else
    return (FALSE);
  end if;                                       -- if match(token_identifier)
end TYPE_DECLARATION;

--------------------------------------------------------------------------------

  -- DISCRIMINANT_PART -->  (DISCRIMINANT_SPECIFICATION
  --                          [; DISCRIMINANT_SPECIFICATION]* )
function DISCRIMINANT_PART return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("DISCRIMINANT_PART");
  end if;
  if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    if (DISCRIMINANT_SPECIFICATION) then
      while (TM.MATCH(TM.TOKEN_SEMICOLON)) loop
        if not (DISCRIMINANT_SPECIFICATION) then
          P4.SYNTAX_ERROR("Discriminant part");
```

```
        end if;                       -- if not discriminant_specification
      end loop;
      if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Discriminant part");
      end if;                         -- if match(token_right_paren)
    else
      P4.SYNTAX_ERROR("Discriminant part");
    end if;                           -- if discriminant_specification
  else
    return (FALSE);
  end if;                             -- if match(token_left_paren)
end DISCRIMINANT_PART;

--------------------------------------------------------------------------------

  -- DISCRIMINANT_SPECIFICATION -->  IDENTIFIER_LIST : NAME [:= EXPRESSION ?]
function DISCRIMINANT_SPECIFICATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("DISCRIMINANT_SPECIFICATION");
  end if;
  if (IDENTIFIER_LIST) then
    if (TM.MATCH(TM.TOKEN_COLON)) then
      if (P3.NAME) then                   -- check for type_mark
        if (TM.MATCH(TM.TOKEN_ASSIGNMENT)) then
          if (P3.EXPRESSION) then
            null;
          else
            P4.SYNTAX_ERROR("Discriminant specification");
          end if;                         -- if expression statement
        end if;                           -- if match(token_assignment)
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Discriminant specification");
      end if;                             -- if name statement
    else
      P4.SYNTAX_ERROR("Discriminant specification");
    end if;                               -- if match(token_colon)
  else
    return (FALSE);
  end if;                                 -- if identifier_list statement
end DISCRIMINANT_SPECIFICATION;

--------------------------------------------------------------------------------

  -- TYPE_DEFINITION -->  ENUMERATION_TYPE_DEFINITION
  --                 -->  INTEGER_TYPE_DEFINITION
  --                 -->  digits FLOATING_OR_FIXED_POINT_CONSTRAINT
                     -->  delta FLOATING_OR_FIXED_POINT_CONSTRAINT
```

```
  --                  -->  array ARRAY_TYPE_DEFINITION
  --                  -->  record RECORD_TYPE_DEFINITION
  --                  -->  access SUBTYPE_INDICATION
  --                  -->  new SUBTYPE_INDICATION
function TYPE_DEFINITION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("TYPE_DEFINITION");
  end if;
  if (P4.ENUMERATION_TYPE_DEFINITION) then
    return (TRUE);
  elsif (P3.INTEGER_TYPE_DEFINITION) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_DIGITS)) or else (TM.MATCH(TM.TOKEN_DELTA)) then
    if (P3.FLOATING_OR_FIXED_POINT_CONSTRAINT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Type definition");
    end if;                           -- floating_or_fixed_point_constraint
  elsif (TM.MATCH(TM.TOKEN_ARRAY)) then
    if (P3.ARRAY_TYPE_DEFINITION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Type definition");
    end if;                                 -- if array_type_definition
  elsif (TM.MATCH(TM.TOKEN_RECORD_STRUCTURE)) then
    if (RECORD_TYPE_DEFINITION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Type definition");
    end if;                                 -- if record_type_definition
  elsif (TM.MATCH(TM.TOKEN_ACCESS)) or else (TM.MATCH(TM.TOKEN_NEW)) then
    if (P3.SUBTYPE_INDICATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Type definition");
    end if;                                 -- if subtype_indication
  else
    return (FALSE);
  end if;
end TYPE_DEFINITION;

-------------------------------------------------------------------------------

  -- RECORD_TYPE_DEFINITION -->  COMPONENT_LIST end record
function RECORD_TYPE_DEFINITION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("RECORD_TYPE_DEFINITION");
  end if;
  if (COMPONENT_LIST) then
```

```
    if (TM.MATCH(TM.TOKEN_END)) then
      if (TM.MATCH(TM.TOKEN_RECORD_STRUCTURE)) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Record type definition");
      end if;                                 -- if match(token_record-structure)
    else
      P4.SYNTAX_ERROR("Record type definition");
    end if;                                   -- if match(token_end)
  else
    return (FALSE);
  end if;                                     -- if component_list statement
end RECORD_TYPE_DEFINITION;

--------------------------------------------------------------------------------

  -- COMPONENT_LIST -->  [COMPONENT_DECLARATION]* [VAARIANT_PART ?]
  --                -->  null ;
function COMPONENT_LIST return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("COMPONENT_LIST");
  end if;
  while (COMPONENT_DECLARATION) loop
    null;
  end loop;
  if (VARIANT_PART) then
    null;
  elsif (TM.MATCH(TM.TOKEN_NULL)) then
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      null;
    end if;
  end if;
  return (TRUE);
end COMPONENT_LIST;

--------------------------------------------------------------------------------

  -- COMPONENT_DECLARATION -->  IDENTIFIER_LIST : SUBTYPE_INDICATION
  --                              [:= EXPRESSION ?] ;
function COMPONENT_DECLARATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("COMPONENT_DECLARATION");
  end if;
  if (IDENTIFIER_LIST) then
    if (TM.MATCH(TM.TOKEN_COLON)) then
      if (P3.SUBTYPE_INDICATION) then
        if (TM.MATCH(TM.TOKEN_ASSIGNMENT)) then
          if (P3.EXPRESSION) then
            if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
```

```
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Component declaration");
        end if;                        -- if match(token_semicolon)
      else
        P4.SYNTAX_ERROR("Component declaration");
      end if;                          -- if expression statement
    end if;                            -- if match(token_assignment)
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Component declaration");
    end if;                            -- if match(token_semicolon)
   else
    P4.SYNTAX_ERROR("Component declaration");
   end if;                             -- if subtype_indication statement
  else
   P4.SYNTAX_ERROR("Component declaration");
  end if;                              -- if match(token_colon)
 else
  return (FALSE);
 end if;                               -- if identifier_list statement
end COMPONENT_DECLARATION;

--------------------------------------------------------------------------------

 -- VARIANT_PART -->  case identifier is [VARIANT]+ end case ;
function VARIANT_PART return boolean is
begin
 if (P4.PRINT_CALLS) then
  P4.OUT_PUT("VARIANT_PART");
 end if;
 if (TM.MATCH(TM.TOKEN_CASE)) then
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
   if (TM.MATCH(TM.TOKEN_IS)) then
    if (VARIANT) then
     while (VARIANT) loop
      null;
     end loop;
     if (TM.MATCH(TM.TOKEN_END)) then
      if (TM.MATCH(TM.TOKEN_CASE)) then
       if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        return (TRUE);
       else
        P4.SYNTAX_ERROR("Variant part");
       end if;                         -- if match(token_semicolon)
      else
       P4.SYNTAX_ERROR("Variant part");
      end if;                          -- if match(token_case)
     else
      P4.SYNTAX_ERROR("Variant part");
```

```
        end if;                              -- if match(token_end)
      else
        P4.SYNTAX_ERROR("Variant part");
      end if;                                -- if variant statement
    else
      P4.SYNTAX_ERROR("Variant part");
    end if;                                  -- if match(token_is)
  else
    P4.SYNTAX_ERROR("Variant part");
  end if;                                    -- if match(token_identifier)
else
  return (FALSE);
end if;                                      -- if match(token_case)
end VARIANT_PART;
```

```
  -- VARIANT -->  when CHOICE [| CHOICE]* => COMPONENT_LIST
function VARIANT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("VARIANT");
  end if;
  if (TM.MATCH(TM.TOKEN_WHEN)) then
    if (P3.CHOICE) then
      while (TM.MATCH(TM.TOKEN_BAR)) loop
        if not (P3.CHOICE) then
          P4.SYNTAX_ERROR("Variant");
        end if;                              -- if not choice statement
      end loop;
      if (TM.MATCH(TM.TOKEN_ARROW)) then
        if (COMPONENT_LIST) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Variant");
        end if;                              -- if component_list statement
      else
        P4.SYNTAX_ERROR("Variant");
      end if;                                -- if match(token_arrow)
    else
      P4.SYNTAX_ERROR("Variant");
    end if;                                  -- if choice statement
  else
    return (FALSE);
  end if;                                    -- if match(token_when)
end VARIANT;
```

```
  -- WITH_OR_USE_CLAUSE -->  identifier [, identifier]* ;
function WITH_OR_USE_CLAUSE return boolean is
```

```
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("WITH_OR_USE_CLAUSE");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    while (TM.MATCH(TM.TOKEN_COMMA)) loop
      if not (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
        P4.SYNTAX_ERROR("With or use clause");
      end if;
    end loop;
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("With or use clause");
    end if;                                   -- if match(token_semicolon)
  else
    return (FALSE);
  end if;                                     -- if match(token_identifier)
end WITH_OR_USE_CLAUSE;

--------------------------------------------------------------------------------

  -- FORMAL_PART -->  (PARAMETER_SPECIFICATION [; PARAMETER_SPECIFICATION]* )
function FORMAL_PART return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("FORMAL_PART");
  end if;
  if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    if (PARAMETER_SPECIFICATION) then
      while (TM.MATCH(TM.TOKEN_SEMICOLON)) loop
        if not (PARAMETER_SPECIFICATION) then
          P4.SYNTAX_ERROR("Formal part");
        end if;                               -- if not parameter_specification
      end loop;
      if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Formal part");
      end if;                                 -- if match(token_right_paren)
    else
      P4.SYNTAX_ERROR("Formal part");
    end if;                                   -- if parameter_specification
  else
    return (FALSE);
  end if;                                     -- if match(token_left_paren)
end FORMAL_PART;

--------------------------------------------------------------------------------
```

```
  -- IDENTIFIER_DECLARATION -->  IDENTIFIER_LIST : IDENTIFIER_DECLARATION_TAIL
function IDENTIFIER_DECLARATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("IDENTIFIER_DECLARATION");
  end if;
  if (IDENTIFIER_LIST) then
    if (TM.MATCH(TM.TOKEN_COLON)) then
      if (IDENTIFIER_DECLARATION_TAIL) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Identifier declaration");
      end if;                              -- if identifier_list
    else
      P4.SYNTAX_ERROR("Identifier declaration");
    end if;                          -- if match(token_colon)
  else
    return(FALSE);
  end if;                            -- if identifier_list
end IDENTIFIER_DECLARATION;

--------------------------------------------------------------------------------

  -- IDENTIFIER_DECLARATION_TAIL -->  exception EXCEPTION_TAIL
  --                             -->  constant CONSTANT_TERM
  --                             -->  array ARRAY_TYPE_DEFINITION
  --                                    [:= EXPRESSION ?] ;
  --                             -->  NAME  IDENTIFIER_TAIL
function IDENTIFIER_DECLARATION_TAIL return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("IDENTIFIER_DECLARATION_TAIL");
  end if;
  if (TM.MATCH(TM.TOKEN_EXCEPTION)) then
    if (EXCEPTION_TAIL) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Identifier declaration tail");
    end if;                                -- if exception tail statement
  elsif (TM.MATCH(TM.TOKEN_CONSTANT)) then
    if (CONSTANT_TERM) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Identifier declaration tail");
    end if;                                -- if constant_term statement
  elsif (TM.MATCH(TM.TOKEN_ARRAY)) then
    if (P3.ARRAY_TYPE_DEFINITION) then
      if (TM.MATCH(TM.TOKEN_ASSIGNMENT)) then
        if (P3.EXPRESSION) then
          null;
        else
```

```
        P4.SYNTAX_ERROR("Identifier declaration tail");
      end if;                              -- if expression statement
    end if;                                -- if match(token_assignment)
  else
    P4.SYNTAX_ERROR("Identifier declaration tail");
  end if;                                  -- if array_type_definition
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    P4.SYNTAX_ERROR("Identifier declaration tail");
  end if;                                  -- if match(token_semicolon)
elsif (P3.NAME) then
  if (IDENTIFIER_TAIL) then
    return (TRUE);
  else
    P4.SYNTAX_ERROR("Identifier declaration tail");
  end if;                                  -- if identifier_tail
else
  return (FALSE);
end if;                                    -- if match(token_exception)
end IDENTIFIER_DECLARATION_TAIL;

--------------------------------------------------------------------------------

  -- EXCEPTION_TAIL -->  ;
  --                -->  renames NAME ;
function EXCEPTION_TAIL return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("EXCEPTION_TAIL");
  end if;
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_RENAMES)) then
    if (P3.NAME) then
      if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Exception tail");
      end if;                              -- if match(token_semicolon)
    else
      P4.SYNTAX_ERROR("Exception tail");
    end if;                                -- if name statement
  else
    return (FALSE);
  end if;                                  -- if match(token_semicolon)
end EXCEPTION_TAIL;

--------------------------------------------------------------------------------
```

```
  -- EXCEPTION_CHOICE -->  NAME
  --                  -->  others
function EXCEPTION_CHOICE return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("EXCEPTION_CHOICE");
  end if;
  if (P3.NAME) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_OTHERS)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end EXCEPTION_CHOICE;

--------------------------------------------------------------------------------

  -- CONSTANT_TERM -->  array ARRAY_TYPE_DEFINITION [:= EXPRESSION ?] ;
  --               -->  := EXPRESSION ;
  --               -->  NAME  IDENTIFIER_TAIL
function CONSTANT_TERM return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("CONSTANT_TERM");
  end if;
  if (TM.MATCH(TM.TOKEN_ARRAY)) then
    if (P3.ARRAY_TYPE_DEFINITION) then
      if (TM.MATCH(TM.TOKEN_ASSIGNMENT)) then
        if (P3.EXPRESSION) then
          null;
        else
          P4.SYNTAX_ERROR("Constant term");
        end if;                            -- if expression statement
      end if;                              -- if match(token_assignment)
    else
      P4.SYNTAX_ERROR("Constant term");
    end if;                                -- if array_type_definition
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Constant term");
    end if;                                -- if match(token_semicolon)
  elsif (TM.MATCH(TM.TOKEN_ASSIGNMENT)) then
    if (P3.EXPRESSION) then
      if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Constant term");
      end if;                              -- if match(token_semicolon)
    else
```

```
      P4.SYNTAX_ERROR("Constant term");
    end if;                                     -- if expression statement
  elsif (P3.NAME) then
    if (IDENTIFIER_TAIL) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Constant term");
    end if;                                     -- if identifier_tail statement
  else
    return (FALSE);
  end if;                                       -- if match(token_array)
end CONSTANT_TERM;

--------------------------------------------------------------------------------

  -- IDENTIFIER_TAIL -->  [CONSTRAINT ?] [:= EXPRESSION ?] ;
  --                 -->  [renames NAME ?] ;
function IDENTIFIER_TAIL return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("IDENTIFIER_TAIL");
  end if;
  if (P3.CONSTRAINT) then
    null;
  end if;                                       -- if constraint statement
  if (TM.MATCH(TM.TOKEN_RENAMES)) then
    if (P3.NAME) then
      null;
    else
      P4.SYNTAX_ERROR("Identifier tail");
    end if;                                     -- if name statement
  end if;                                       -- if match(token_renames)
  if (TM.MATCH(TM.TOKEN_ASSIGNMENT)) then
    if (P3.EXPRESSION) then
      null;
    else
      P4.SYNTAX_ERROR("Identifier tail");
    end if;                                     -- if expression statement
  end if;                                       -- if match(token_assignment)
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE);
  end if;                                       -- if match(token_semicolon)
end IDENTIFIER_TAIL;

--------------------------------------------------------------------------------

  -- PARAMETER_SPECIFICATION -->  IDENTIFIER_LIST : MODE  NAME [:= EXPRESSION ?]
function PARAMETER_SPECIFICATION return boolean is
begin
```

```
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("PARAMETER_SPECIFICATION");
  end if;
  if (IDENTIFIER_LIST) then
    if (TM.MATCH(TM.TOKEN_COLON)) then
      if (MODE) then
        if (P3.NAME) then                     -- check for type_mark
          if (TM.MATCH(TM.TOKEN_ASSIGNMENT)) then
            if (P3.EXPRESSION) then
              null;
            else
              P4.SYNTAX_ERROR("Parameter specification");
            end if;                           -- if expression statement
          end if;                             -- if match(token_assignment)
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Parameter specification");
        end if;                               -- if name statement
      else
        P4.SYNTAX_ERROR("Parameter specification");
      end if;                                 -- if mode statement
    else
      P4.SYNTAX_ERROR("Parameter specification");
    end if;                                   -- if match(token_colon)
  else
    return (FALSE);
  end if;                                     -- if identifier_list statement
end PARAMETER_SPECIFICATION;

--------------------------------------------------------------------------------

 -- IDENTIFIER_LIST -->  identifier [, identifier]*
function IDENTIFIER_LIST return boolean is
TEMP_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION : natural;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("IDENTIFIER_LIST");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    LOCATION := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    TM.MATCHED_TOKEN(TEMP_TOKEN);
    SYMBOL_TABLE.INSERT_SYM_TAB(TEMP_TOKEN.LEXEME(1..TEMP_TOKEN.LEXEME_SIZE),
                            SYMBOL_TABLE.OBJECT_DECLARATION_TAG, LOCATION);
    while (TM.MATCH(TM.TOKEN_COMMA)) loop
      if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
      TM.MATCHED_TOKEN(TEMP_TOKEN);
      SYMBOL_TABLE.INSERT_SYM_TAB(TEMP_TOKEN.LEXEME(1..TEMP_TOKEN.LEXEME_SIZE),
                              SYMBOL_TABLE.OBJECT_DECLARATION_TAG,
                              LOCATION);
      else
```

```
      P4.SYNTAX_ERROR("Identifier list");
    end if;                     -- if not match(token_identifer) statement
  end loop;
  return (TRUE);
else
  return (FALSE);
end if;                         -- if match(token_identifier) statement
end IDENTIFIER_LIST;

--------------------------------------------------------------------------------

  -- MODE -->  [in ?]
  --      -->  in out
  --      -->  out
function MODE return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("MODE");
  end if;
  if (TM.MATCH(TM.TOKEN_IN)) then
    if (TM.MATCH(TM.TOKEN_OUT)) then
      null;
    end if;
  elsif (TM.MATCH(TM.TOKEN_OUT)) then
    null;
  end if;
  return (TRUE);
end MODE;

--------------------------------------------------------------------------------

  -- DESIGNATOR -->  identifier
  --            -->  string_literal
function DESIGNATOR return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("DESIGNATOR");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_STRING_LITERAL)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end DESIGNATOR;

--------------------------------------------------------------------------------

  -- SIMPLE_STATEMENT -->  null ;
  --                  -->  ASSIGNMENT_OR_PROCEDURE_CALL
```

```
  --                    -->  exit EXIT_STATEMENT
  --                    -->  return RETURN_STATEMENT
  --                    -->  goto GOTO_STATEMENT
  --                    -->  delay DELAY_STATEMENT
  --                    -->  abort ABORT_STATEMENT
  --                    -->  raise RAISE_STATEMENT
function SIMPLE_STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SIMPLE_STATEMENT");
  end if;
  if (TM.MATCH(TM.TOKEN_NULL)) then
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Simple statement");
    end if;
  elsif (ASSIGNMENT_OR_PROCEDURE_CALL) then -- includes a check for a
    return (TRUE);                          -- code statement and an
  elsif (TM.MATCH(TM.TOKEN_EXIT)) then      -- entry call statement.
    if (P3.EXIT_STATEMENT) then
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Simple statement");
    end if;
  elsif (TM.MATCH(TM.TOKEN_RETURN)) then
    if (P3.RETURN_STATEMENT) then
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Simple statement");
    end if;
  elsif (TM.MATCH(TM.TOKEN_GOTO)) then
    if (P3.GOTO_STATEMENT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Simple statement");
    end if;
  elsif (TM.MATCH(TM.TOKEN_DELAY)) then
    if (P3.DELAY_STATEMENT) then
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Simple statement");
    end if;
  elsif (TM.MATCH(TM.TOKEN_ABORT)) then
    if (P3.ABORT_STATEMENT) then
      CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      return (TRUE);
```

```
    else
      P4.SYNTAX_ERROR("Simple statement");
    end if;
  elsif (TM.MATCH(TM.TOKEN_RAISE)) then
    if (P3.RAISE_STATEMENT) then
      COOE_BLOCKER.INCREMENT_STATEMENT_COUNT;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Simple statement");
    end if;
  else
    return (FALSE);
  end if;
end SIMPLE_STATEMENT;

-----------------------------------------------------------------------------

  -- ASSIGNMENT_OR_PROCEDURE_CALL -->  NAME := EXPRESSION ;
  --                              -->  NAME ;
function ASSIGNMENT_OR_PROCEOURE_CALL return boolean is
SEARCH_POINTER : SYMBOL_TABLE.SYM_TAB_ACCESS;
SEARCH_TOKEN   : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE   : positive;
use SYMBOL_TABLE;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("ASSIGNMENT_OR_PROCEDURE_CALL");
  end if;
  SYMBOL_TABLE.SAVE_CURRENT_ENTRY;
  if (P3.NAME) then
    if (TM.MATCH(TM.TOKEN_ASSIGNMENT)) then
      if (P3.EXPRESSION) then
        TM.MATCHEO_TOKEN(SEARCH_TOKEN);
        SEARCH_POINTER := SYMBOL_TABLE.RETRIEVE_SYM;
        if ((SEARCH_POINTER /= null) and then
        (SEARCH_POINTER.TAG_TYPE = SYMBOL_TABLE.FUNCTION_DECLARATION_TAG)) then
          LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
          COOE_BLOCKER.INCREMENT_STATEMENT_COUNT;
          COOE_BLOCKER.EXIT_COOE_BLOCK(SEARCH_TOKEN.SOURCE);
          NET_GENERATOR.CALL(LOCATION_ONE, SEARCH_POINTER);
          COOE_BLOCKER.ENTER_CODE_BLOCK(SEARCH_TOKEN.SOURCE, "");
        else
          CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
        end if;
        if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
          SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
          return (TRUE);                    -- parsed an assignment statement
        else
          P4.SYNTAX_ERROR("Assignment or procedure call");
        end if;                             -- if match(token_semicolon)
      else
```

```
        P4.SYNTAX_ERROR("Assignment or procedure call");
      end if;                                   -- if expression statement
    elsif (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      TM.MATCHED_TOKEN(SEARCH_TOKEN);
      SEARCH_POINTER := SYMBOL_TABLE.RETRIEVE_SYM;
      if ((SEARCH_POINTER /= null) and then
      (SEARCH_POINTER.TAG_TYPE = SYMBOL_TABLE.PROCEDURE_DECLARATION_TAG)) then
        LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
        CODE_BLOCKER.EXIT_CODE_BLOCK(SEARCH_TOKEN.SOURCE);
        NET_GENERATOR.CALL(LOCATION_ONE, SEARCH_POINTER);
        CODE_BLOCKER.ENTER_CODE_BLOCK(SEARCH_TOKEN.SOURCE, "");
      elsif ((SEARCH_POINTER /= null) and then
      (SEARCH_POINTER.TAG_TYPE = SYMBOL_TABLE.ENTRY_TAG)) then
        LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
        CODE_BLOCKER.EXIT_CODE_BLOCK(SEARCH_TOKEN.SOURCE);
        NET_GENERATOR.ENTRY_CALL(LOCATION_ONE, SEARCH_POINTER);
        CODE_BLOCKER.ENTER_CODE_BLOCK(SEARCH_TOKEN.SOURCE, "");
      end if;
      SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
      return (TRUE);                   -- parsed a procedure call statement
    else
      P4.SYNTAX_ERROR("Assignment or procedure call");
    end if;                                     -- if match(token_assignment)
  else
    SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
    return (FALSE);
  end if;                                       -- if name statement
end ASSIGNMENT_OR_PROCEDURE_CALL;

-------------------------------------------------------------------------------

  -- LABEL -->  << identifier >>
function LABEL return boolean is
START_TOKEN  : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE : positive;
LOCATION_TWO : positive;
use SYMBOL_TABLE;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("LABEL");
  end if;
  if (TM.MATCH(TM.TOKEN_LEFT_BRACKET)) then
    if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
      TM.MATCHED_TOKEN(START_TOKEN);
      if (TM.MATCH(TM.TOKEN_RIGHT_BRACKET)) then
        if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT /= 0) then
          LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
          CODE_BLOCKER.EXIT_CODE_BLOCK(START_TOKEN.SOURCE);
          CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "LABELLED BLOCK");
```

```
        LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
        NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
      else
        CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
        CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "LABELLED BLOCK");
        CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
        LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
      end if;
      if (SYMBOL_TABLE.FIND_KEY(START_TOKEN.LEXEME(1..
      START_TOKEN.LEXEME_SIZE)) = null) then
        SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.
                                    LEXEME(1..START_TOKEN.LEXEME_SIZE),
                                    SYMBOL_TABLE.LABEL_NAME, LOCATION_TWO);
      else
        SYMBOL_TABLE.UPDATE_SYM_TAB(LOCATION_TWO);
      end if;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Label");
    end if;                                  -- if match(token_right_bracket)
  else
    P4.SYNTAX_ERROR("Label");
  end if;                                    -- if match(token_identifier)
else
  return (FALSE);
end if;                                      -- if match(token_left_bracket)
end LABEL;

--------------------------------------------------------------------------------

  -- ENTRY_DECLARATION -->  entry identifier [(DISCRETE_RANGE) ?]
  --                         [FORMAL_PART ?] ;
function ENTRY_DECLARATION return boolean is
START_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("ENTRY_DECLARATION");
  end if;
  if (TM.MATCH(TM.TOKEN_ENTRY)) then
    if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
      TM.MATCHED_TOKEN(START_TOKEN);
      SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.LEXEME(1..
        START_TOKEN.LEXEME_SIZE), SYMBOL_TABLE.ENTRY_TAG, 0);
      SYMBOL_TABLE.INSERT_SYM_TAB("END", SYMBOL_TABLE.LABEL_NAME, 0);
      if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
        if (P3.DISCRETE_RANGE) then
          if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
            null;
          else
            P4.SYNTAX_ERROR("Entry declaration");
```

```
        end if;                           -- if match(token_right_paren)
      else
        P4.SYNTAX_ERROR("Entry declaration");
      end if;                             -- if discrete_range statement
    end if;                               -- if match(token_left_paren)
    if (FORMAL_PART) then
      null;
    end if;                               -- if formal_part statement
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      TM.MATCHED_TOKEN(START_TOKEN);
      SYMBOL_TABLE.EXIT_SCOPE;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Entry declaration");
    end if;                               -- if match(token_semicolon)
  else
    P4.SYNTAX_ERROR("Entry declaration");
  end if;                                 -- if match(token_identifier)
 else
   return (FALSE);
 end if;                                  -- if match(token_entry)
end ENTRY_DECLARATION;

-------------------------------------------------------------------------------

 -- REPRESENTATION_CLAUSE -->  for NAME use record RECORD_REPRESENTATION_CLAUSE
 --                       -->  for NAME use [at ?] SIMPLE_EXPRESSION;
function REPRESENTATION_CLAUSE return boolean is
begin
 if (P4.PRINT_CALLS) then
   P4.OUT_PUT("REPRESENTATION_CLAUSE");
 end if;
 if (TM.MATCH(TM.TOKEN_FOR)) then
   if (P3.NAME) then
     if (TM.MATCH(TM.TOKEN_USE)) then
       if (TM.MATCH(TM.TOKEN_RECORD_STRUCTURE)) then
         if (RECORD_REPRESENTATION_CLAUSE) then
           return (TRUE);
         else
           P4.SYNTAX_ERROR("Representation clause");
         end if;                          -- if record_representation_clause
       elsif (TM.MATCH(TM.TOKEN_AT)) then
         if (P3.SIMPLE_EXPRESSION) then
           if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
             return (TRUE);
           else
             P4.SYNTAX_ERROR("Representation clause");
           end if;                        -- if match(token_semicolon)
         else
           P4.SYNTAX_ERROR("Representation clause");
         end if;                          -- if simple_expression statement
```

```
      elsif (P3.SIMPLE_EXPRESSION) then
        if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Representation clause");
        end if;                             -- if match(token_semicolon)
      else
        P4.SYNTAX_ERROR("Representation clause");
      end if;                               -- if match(token_record)
    else
      P4.SYNTAX_ERROR("Representation clause");
    end if;                                 -- if match(token_use)
  else
    P4.SYNTAX_ERROR("Representation clause");
  end if;                                   -- if name statement
 else
  return (FALSE);
 end if;                                    -- if match(token_for)
end REPRESENTATION_CLAUSE;

-------------------------------------------------------------------------------

 -- RECORD_REPRESENTATION_CLAUSE --> [at mod SIMPLE_EXPRESSION ?]
 --                                  [NAME at SIMPLE_EXPRESSION range RANGES]*
 --                                     end record ;
function RECORD_REPRESENTATION_CLAUSE return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("RECORD_REPRESENTATION_CLAUSE");
  end if;
  if (TM.MATCH(TM.TOKEN_AT)) then
    if (TM.MATCH(TM.TOKEN_MOD)) then
      if (P3.SIMPLE_EXPRESSION) then
        null;
      else
        P4.SYNTAX_ERROR("Record representation clause");
      end if;                               -- if simple_expression
    else
      P4.SYNTAX_ERROR("Record representation clause");
    end if;                                 -- if match(token_mod)
  end if;                                   -- if match(token_at)
  while (P3.NAME) loop
    if (TM.MATCH(TM.TOKEN_AT)) then
      if (P3.SIMPLE_EXPRESSION) then
        if (TM.MATCH(TM.TOKEN_RANGE)) then
          if (P3.RANGES) then
            null;
          else
            P4.SYNTAX_ERROR("Record representation clause");
          end if;                           -- if ranges statement
        else
```

```
          P4.SYNTAX_ERROR("Record representation clause");
        end if;                               -- if match(token_range)
      else
        P4.SYNTAX_ERROR("Record representation clause");
      end if;                                 -- if simple_expression
    else
      P4.SYNTAX_ERROR("Record representation clause");
    end if;                                   -- if match(token_at)
  end loop;
  if (TM.MATCH(TM.TOKEN_END)) then
    if (TM.MATCH(TM.TOKEN_RECORD_STRUCTURE)) then
      if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Record representation clause");
      end if;                                 -- if match(token_semicolon)
    else
      P4.SYNTAX_ERROR("Record representation clause");
    end if;                                   -- if match(token_record_structure)
  else
    return (FALSE);
  end if;                                     -- if match(token_end)
end RECORD_REPRESENTATION_CLAUSE;

end PARSER_2;
```

```
--*************************************************************--
--                                                             --
--  TITLE:          ADAFLOW                                    --
--                                                             --
--  MODULE NAME:    PACKAGE PARSER_3                           --
--  FILE NAME:      PARSER3.ADS                                --
--                                                             --
--  DATE CREATED:   20 FEB 88                                  --
--  LAST MODIFIED:  28 APR 88                                  --
--                                                             --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                   --
--                                                             --
--  BASED ON A MODIFIED ADA GRAMMAR DEVELOPED BY:              --
--                  LCDR JEFFREY L. NIEDER, USN                --
--                  LT KARL S. FAIRBANKS, JR., USN             --
--                  LCDR PAUL M. HERZIG, USN                   --
--                                                             --
--  DESCRIPTION:  This package defines the functions           --
--       that make up the baseline productions for a top-down, --
--       recursive descent parser.                             --
--                                                             --
--*************************************************************--

package PARSER_3 is
  function SUBTYPE_INDICATION return boolean;
  function ARRAY_TYPE_DEFINITION return boolean;
  function CHOICE return boolean;
  function ITERATION_SCHEME return boolean;
  function LOOP_PARAMETER_SPECIFICATION return boolean;
  function EXPRESSION return boolean;
  function RELATION return boolean;
  function RELATION_TAIL return boolean;
  function SIMPLE_EXPRESSION return boolean;
  function SIMPLE_EXPRESSION_TAIL return boolean;
  function TERM return boolean;
  function FACTOR return boolean;
  function PRIMARY return boolean;
  function CONSTRAINT return boolean;
  function FLOATING_OR_FIXED_POINT_CONSTRAINT return boolean;
  function INDEX_CONSTRAINT return boolean;
  function RANGES return boolean;
  function AGGREGATE return boolean;
  function COMPONENT_ASSOCIATION return boolean;
  function ALLOCATOR return boolean;
  function NAME return boolean;
  function NAME_TAIL return boolean;
  function LEFT_PAREN_NAME_TAIL return boolean;
  function ATTRIBUTE_DESIGNATOR return boolean;
  function INTEGER_TYPE_DEFINITION return boolean;
  function DISCRETE_RANGE return boolean;
  function EXIT_STATEMENT return boolean;
```

```
  function RETURN_STATEMENT return boolean;
  function GOTO_STATEMENT return boolean;
  function DELAY_STATEMENT return boolean;
  function ABORT_STATEMENT return boolean;
  function RAISE_STATEMENT return boolean;
end PARSER_3;
```

```
--*************************************************************--
--                                                             --
--  TITLE:          ADAFLOW                                    --
--                                                             --
--  MODULE NAME:    PACKAGE PARSER_3                           --
--  FILE NAME:      PARSER3.ADB                                --
--                                                             --
--  DATE CREATED:   20 FEB 88                                  --
--  LAST MODIFIED:  28 APR 88                                  --
--                                                             --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                   --
--                                                             --
--  BASED ON A MODIFIED ADA GRAMMAR DEVELOPED BY:              --
--                  LCDR JEFFREY L. NIEDER, USN                --
--                  LT KARL S. FAIRBANKS, JR., USN             --
--                  LCDR PAUL M. HERZIG, USN                   --
--                                                             --
--  DESCRIPTION:  This package implements the functions        --
--       that make up the baseline productions for a top-down, --
--       recursive descent parser.  Each function is preceded  --
--       by the grammar productions they are implementing.     --
--                                                             --
--*************************************************************--

with PARSER_4, TOKEN_MATCHER, TOKEN_SCANNER, CODE_BLOCKER,
     SYMBOL_TABLE, NET_GENERATOR;

package body PARSER_3 is
  package TM renames TOKEN_MATCHER;
  package P4 renames PARSER_4;

  -- SUBTYPE_INDICATION -->  NAME [CONSTRAINT ?]
function SUBTYPE_INDICATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SUBTYPE_INDICATION");
  end if;
  if (NAME) then                              -- check for type_mark
    if (CONSTRAINT) then
      null;
    end if;
    return (TRUE);
  else
    return (FALSE);
  end if;
end SUBTYPE_INDICATION;

-----------------------------------------------------------------------

  -- ARRAY_TYPE_DEFINITION -->  (INDEX_CONSTRAINT of SUBTYPE_INDICATION
  -- this function parses both constrained and unconstrained arrays
```

```
function ARRAY_TYPE_DEFINITION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("ARRAY_TYPE_DEFINITION");
  end if;
  if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    if (INDEX_CONSTRAINT) then
      if (TM.MATCH(TM.TOKEN_OF)) then
        if (SUBTYPE_INDICATION) then
          return (TRUE);
        else
          P4.SYNTAX_ERROR("Array definition");
        end if;                              -- if subtype_indication
      else
        P4.SYNTAX_ERROR("Array definition");
      end if;                                -- if match(token_of)
    else
      P4.SYNTAX_ERROR("Array definition");
    end if;                                  -- if index_constraint statement
  else
    return (FALSE);
  end if;                                    -- if match(token_left_paren)
end ARRAY_TYPE_DEFINITION;

--------------------------------------------------------------------------------

  -- CHOICE -->  EXPRESSION [..SIMPLE_EXPRESSION ?]
  --        -->  EXPRESSION [CONSTRAINT ?]
  --        -->  others
function CHOICE return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("CHOICE");
  end if;
  if (EXPRESSION) then
    if (TM.MATCH(TM.TOKEN_RANGE_DOTS)) then -- check for discrete_range
      if (SIMPLE_EXPRESSION) then
        null;
      else
        P4.SYNTAX_ERROR("Choice");
      end if;                                -- if simple_expression statement
    elsif (CONSTRAINT) then
      null;
    end if;                                  -- if match token_range_dots
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_OTHERS)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end CHOICE;
```

```
--------------------------------------------------------------------------------

  -- ITERATION_SCHEME -->  while EXPRESSION
  --                   -->  for LOOP_PARAMETER_SPECIFICATION
function ITERATION_SCHEME return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("ITERATION_SCHEME");
  end if;
  if (TM.MATCH(TM.TOKEN_WHILE)) then
    if (EXPRESSION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Iteration scheme");
    end if;
  elsif (TM.MATCH(TM.TOKEN_FOR)) then
    if (LOOP_PARAMETER_SPECIFICATION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Iteration scheme");
    end if;
  else
    return (FALSE);
  end if;
end ITERATION_SCHEME;

--------------------------------------------------------------------------------

  -- LOOP_PARAMETER_SPECIFICATION -->  identifier in [reverse ?] DISCRETE_RANGE
function LOOP_PARAMETER_SPECIFICATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("LOOP_PARAMETER_SPECIFICATION");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    if (TM.MATCH(TM.TOKEN_IN)) then
      if (TM.MATCH(TM.TOKEN_REVERSE)) then
        null;
      end if;                              -- if match(token_reverse)
      if (DISCRETE_RANGE) then
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Loop parameter specification");
      end if;                              -- if discrete_range statement
    else
      P4.SYNTAX_ERROR("Loop parameter specification");
    end if;                                -- if match(token_in)
  else
    return (FALSE);
```

```
  end if;                                    -- if match(token_identifier)
end LOOP_PARAMETER_SPECIFICATION;

--------------------------------------------------------------------------------

  -- EXPRESSION -->  RELATION [RELATION_TAIL ?]
function EXPRESSION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("EXPRESSION");
  end if;
  if (RELATION) then
    if (RELATION_TAIL) then
      null;
    end if;                                  -- if relation_tail statement
    return (TRUE);
  else
    return (FALSE);
  end if;                                    -- if relation statement
end EXPRESSION;

--------------------------------------------------------------------------------

  -- RELATION -->  SIMPLE_EXPRESSION [SIMPLE_EXPRESSION_TAIL ?]
function RELATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("RELATION");
  end if;
  if (SIMPLE_EXPRESSION) then
    if (SIMPLE_EXPRESSION_TAIL) then
      null;
    end if;                            -- if simple_expression_tail statement
    return (TRUE);
  else
    return (FALSE);
  end if;                                    -- if simple_expression statement
end RELATION;

--------------------------------------------------------------------------------

  -- RELATION_TAIL -->  [and [then ?] RELATION]*
  --               -->  [or [else ?] RELATION]*
  --               -->  [xor RELATION]*
function RELATION_TAIL return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("RELATION_TAIL");
  end if;
  while (TM.MATCH(TM.TOKEN_AND)) loop
    if (TM.MATCH(TM.TOKEN_THEN)) then
```

```
      null;
    end if;                                   -- if match(token_then)
    if not (RELATION) then
      P4.SYNTAX_ERROR("Relation tail");
    end if;                                   -- if not relation statement
  end loop;
  while (TM.MATCH(TM.TOKEN_OR)) loop
    if (TM.MATCH(TM.TOKEN_ELSE)) then
      null;
    end if;                                   -- if match(token_else)
    if not (RELATION) then
      P4.SYNTAX_ERROR("Relation tail");
    end if;                                   -- if not relation statement
  end loop;
  while (TM.MATCH(TM.TOKEN_XOR)) loop
    if not (RELATION) then
      P4.SYNTAX_ERROR("Relation tail");
    end if;                                   -- if not relation statement
  end loop;
  return (TRUE);
end RELATION_TAIL;

--------------------------------------------------------------------------------

  -- SIMPLE_EXPRESSION -->  [+ ?] TERM [BINARY_ADDING_OPERATOR  TERM]*
  --                   -->  [- ?] TERM [BINARY ADDING_OPERATOR  TERM]*
function SIMPLE_EXPRESSION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SIMPLE_EXPRESSION");
  end if;
  if (TM.MATCH(TM.TOKEN_PLUS) or TM.MATCH(TM.TOKEN_MINUS)) then
    if (TERM) then
      while (P4.BINARY_ADDING_OPERATOR) loop
        if not (TERM) then
          P4.SYNTAX_ERROR("Simple expression");
        end if;                               -- if not term statement
      end loop;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Simple expression");
    end if;                                   -- if term statement
  elsif (TERM) then
    while (P4.BINARY_ADDING_OPERATOR) loop
      if not (TERM) then
        P4.SYNTAX_ERROR("Simple expression");
      end if;                                 -- if not term statement
    end loop;
    return (TRUE);
  else
    return (FALSE);
```

```
  end if;                               -- if match(token_plus) et al statement
end SIMPLE_EXPRESSION;

------------------------------------------------------------------------------

  -- SIMPLE_EXPRESSION_TAIL -->  RELATIONAL_OPERATOR  SIMPLE_EXPRESSION
  --                        -->  [not ?] in RANGES
  --                        -->  [not ?] in NAME
function SIMPLE_EXPRESSION_TAIL return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("SIMPLE_EXPRESSION_TAIL");
  end if;
  if (P4.RELATIONAL_OPERATOR) then
    if (SIMPLE_EXPRESSION) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Simple expression tail");
    end if;                                 -- if simple_expression statement
  elsif (TM.MATCH(TM.TOKEN_NOT)) then
    if (TM.MATCH(TM.TOKEN_IN)) then
      if (RANGES) then
        return (TRUE);
      elsif (NAME) then                     -- check for type_mark
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Simple expression tail");
      end if;                               -- if ranges statement
    else
      P4.SYNTAX_ERROR("Simple expression tail");
    end if;                                 -- if match(token_in) statement
  elsif (TM.MATCH(TM.TOKEN_IN)) then
    if (RANGES) then
      return (TRUE);
    elsif (NAME) then                       -- check for type_mark
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Simple expression tail");
    end if;                                 -- if ranges statement
  else
    return (FALSE);
  end if;                                   -- if relational_operator statement
end SIMPLE_EXPRESSION_TAIL;

------------------------------------------------------------------------------

  -- TERM -->  FACTOR [MULTIPLYING_OPERATOR  FACTOR]*
function TERM return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("TERM");
```

```
  end if;
  if (FACTOR) then
    while (P4.MULTIPLYING_OPERATOR) loop
      if not (FACTOR) then
        P4.SYNTAX_ERROR("Term");
      end if;                                   -- if not factor statement
    end loop;
    return (TRUE);
  else
    return (FALSE);
  end if;                                       -- if factor statement
end TERM;

-----------------------------------------------------------------------------

  -- FACTOR -->  PRIMARY [** PRIMARY ?]
  --        -->  abs PRIMARY
  --        -->  not PRIMARY
function FACTOR return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("FACTOR");
  end if;
  if (PRIMARY) then
    if (TM.MATCH(TM.TOKEN_EXPONENT)) then
      if (PRIMARY) then
        null;
      else
        P4.SYNTAX_ERROR("Factor");
      end if;                                   -- if primary statement
    end if;                                     -- if match(token_exponent)
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_ABSOLUTE)) then
    if (PRIMARY) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Factor");
    end if;                                     -- if primary(abs)
  elsif (TM.MATCH(TM.TOKEN_NOT)) then
    if (PRIMARY) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Factor");
    end if;                                     -- if primary(not)
  else
    return (FALSE);
  end if;                                       -- if primary statement
end FACTOR;

-----------------------------------------------------------------------------
```

```
  -- PRIMARY -->  numeric_literal
  --         -->  null
  --         -->  string_literal
  --         -->  new ALLOCATOR
  --         -->  NAME
  --         -->  AGGREGATE
function PRIMARY return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("PRIMARY");
  end if;
  if (TM.MATCH(TM.TOKEN_NUMERIC_LITERAL)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_NULL)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_STRING_LITERAL)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_NEW)) then
    if (ALLOCATOR) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Primary");
    end if;                                     -- if allocator statement
  elsif (NAME) then
    return (TRUE);
  elsif (AGGREGATE) then
    return (TRUE);
  else
    return (FALSE);
  end if;                                       -- if match(token_left_paren)
end PRIMARY;

------------------------------------------------------------------------------

  -- CONSTRAINT -->  range RANGES
  --            -->  range <>
  --            -->  digits FLOATING_OR_FIXED_POINT_CONSTRAINT
  --            -->  delta FLOATING_OR_FIXED_POINT_CONSTRAINT
  --            -->  (INDEX_CONSTRAINT
function CONSTRAINT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("CONSTRAINT");
  end if;
  if (TM.MATCH(TM.TOKEN_RANGE)) then
    if (RANGES) then
      return (TRUE);
    elsif (TM.MATCH(TM.TOKEN_BRACKETS)) then -- check for <> when parsing
      return (TRUE);                          -- an unconstrained array
    else
      P4.SYNTAX_ERROR("Constraint");
```

```
    end if;                                     -- if ranges statement
  elsif (TM.MATCH(TM.TOKEN_DIGITS)) or else (TM.MATCH(TM.TOKEN_DELTA)) then
    if (FLOATING_OR_FIXED_POINT_CONSTRAINT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Constraint");
    end if;
  elsif (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    if (INDEX_CONSTRAINT) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Constraint");
    end if;
  else
    return (FALSE);
  end if;
end CONSTRAINT;

--------------------------------------------------------------------------------

  -- FLOATING_OR_FIXED_POINT_CONSTRAINT -->  SIMPLE_EXPRESSION [range RANGES ?]
function FLOATING_OR_FIXED_POINT_CONSTRAINT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("FLOATING_OR_FIXED_POINT_CONSTRAINT");
  end if;
  if (SIMPLE_EXPRESSION) then
    if (TM.MATCH(TM.TOKEN_RANGE)) then
      if (RANGES) then
        null;
      else
        P4.SYNTAX_ERROR("Floating or fixed point constraint");
      end if;                                   -- if ranges statement
    end if;                                     -- if match(token_range)
    return (TRUE);
  else
    return (FALSE);
  end if;                                       -- if simple_expression statement
end FLOATING_OR_FIXED_POINT_CONSTRAINT;

--------------------------------------------------------------------------------

  -- INDEX_CONSTRAINT -->  DISCRETE_RANGE [, DISCRETE_RANGE]* )
function INDEX_CONSTRAINT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("INDEX_CONSTRAINT");
  end if;
  if (DISCRETE_RANGE) then
    while (TM.MATCH(TM.TOKEN_COMMA)) loop
      if not (DISCRETE_RANGE) then
```

```
      P4.SYNTAX_ERROR("Index constraint");
    end if;                                   -- if not discrete_range
  end loop;
  if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
    return (TRUE);
  else
    P4.SYNTAX_ERROR("Index constraint");
  end if;                                     -- if match(token_right_paren)
else
  return (FALSE);
end if;                                       -- if discrete_range statement
end INDEX_CONSTRAINT;

-------------------------------------------------------------------------------

  -- RANGES -->  SIMPLE_EXPRESSION [..SIMPLE_EXPRESSION ?]
function RANGES return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("RANGES");
  end if;
  if (SIMPLE_EXPRESSION) then
    if (TM.MATCH(TM.TOKEN_RANGE_DOTS)) then
      if (SIMPLE_EXPRESSION) then
        null;
      else
        P4.SYNTAX_ERROR("Ranges");
      end if;                                 -- if simple_expression statement
    end if;                                   -- if match(token_range_dots)
    return (TRUE);
  else
    return (FALSE);
  end if;                                     -- if simple_expression statement
end RANGES;

-------------------------------------------------------------------------------

  -- AGGREGATE -->  (COMPONENT_ASSOCIATION [, COMPONENT_ASSOCIATION]* )
function AGGREGATE return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("AGGREGATE");
  end if;
  if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    if (COMPONENT_ASSOCIATION) then
      while (TM.MATCH(TM.TOKEN_COMMA)) loop
        if not (COMPONENT_ASSOCIATION) then
          P4.SYNTAX_ERROR("Aggregate");
        end if;                               -- if not component association
      end loop;
      if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
```

```
        return (TRUE);
      else
        P4.SYNTAX_ERROR("Aggregate");
      end if;                                    -- if match(token_right_paren)
    else
      P4.SYNTAX_ERROR("Aggregate");
    end if;                                      -- if component_association
  else
    return (FALSE);
  end if;                                        -- if match(token_left_paren)
end AGGREGATE;

--------------------------------------------------------------------------------

  -- COMPONENT_ASSOCIATION -->  [CHOICE [| CHOICE]* => ?] EXPRESSION
function COMPONENT_ASSOCIATION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("COMPONENT_ASSOCIATION");
  end if;
  if (CHOICE) then
    while (TM.MATCH(TM.TOKEN_BAR)) loop
      if not (CHOICE) then
        P4.SYNTAX_ERROR("Component asociation");
      end if;
    end loop;
    if (TM.MATCH(TM.TOKEN_ARROW)) then
      if (EXPRESSION) then
        null;
      else
        P4.SYNTAX_ERROR("Component asociation");
      end if;                                    -- if expression statement
    end if;                                      -- if match(token_arrow)
    return (TRUE);
  else
    return (FALSE);
  end if;                                        -- if choice statement
end COMPONENT_ASSOCIATION;

--------------------------------------------------------------------------------

  -- ALLOCATOR -->  SUBTYPE_INDICATION ['AGGREGATE ?]
function ALLOCATOR return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("ALLOCATOR");
  end if;
  if (SUBTYPE_INDICATION) then
    if (TM.MATCH(TM.TOKEN_APOSTROPHE)) then
      if (AGGREGATE) then
        null;
```

```
      else
        P4.SYNTAX_ERROR("Allocator");
      end if;                                      -- if aggregate statement
    end if;                                        -- if match(token_apostrophe)
    return (TRUE);
  else
    return (FALSE);
  end if;                                          -- if subtype_indication statement
end ALLOCATOR;

--------------------------------------------------------------------------------

  -- NAME -->  identifier [NAME_TAIL ?]
  --      -->  character_literal [NAME_TAIL ?]
  --      -->  string_literal [NAME_TAIL ?]
function NAME return boolean is
SEARCH_POINTER : SYMBOL_TABLE.SYM_TAB_ACCESS;
START_TOKEN    : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE   : positive;
LOCATION_TWO   : positive;
use SYMBOL_TABLE;
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("NAME");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    TM.MATCHED_TOKEN(START_TOKEN);
    SEARCH_POINTER :=
      SYMBOL_TABLE.FIND_KEY(START_TOKEN.LEXEME(1..START_TOKEN.LEXEME_SIZE));
    if (NAME_TAIL) then
      null;
    elsif (TM.MATCH(TM.TOKEN_COLON)) then
      if (CODE_BLOCKER.CURRENT_STATEMENT_COUNT /= 0) then
        LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        CODE_BLOCKER.EXIT_CODE_BLOCK(START_TOKEN.SOURCE);
        CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "LABELLED BLOCK");
        LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
        CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
        NET_GENERATOR.CONNECT_BLOCKS(LOCATION_ONE, LOCATION_TWO);
      else
        CODE_BLOCKER.DELETE_CODE_BLOCK_ENTER;
        CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "LABELLED BLOCK");
        CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
        LOCATION_TWO := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
      end if;
      if (SYMBOL_TABLE.FIND_KEY(START_TOKEN.LEXEME(1..
      START_TOKEN.LEXEME_SIZE)) = null) then
        SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.
                                    LEXEME(1..START_TOKEN.LEXEME_SIZE),
                                    SYMBOL_TABLE.LABEL_NAME, LOCATION_TWO);
      else
```

```
        SYMBOL_TABLE.UPDATE_SYM_TAB(LOCATION_TWO);
      end if;
      return (FALSE);
    end if;
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_CHARACTER_LITERAL)) then
    if (NAME_TAIL) then
      null;
    end if;
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_STRING_LITERAL)) then
    if (NAME_TAIL) then
      null;
    end if;
    return (TRUE);
  else
    return (FALSE);
  end if;
end NAME;

--------------------------------------------------------------------------------

  -- NAME_TAIL -->  (LEFT_PAREN_NAME_TAIL
  --           -->  .SELECTOR [NAME_TAIL]*
  --           -->  'AGGREGATE [NAME_TAIL]*
  --           -->  'ATTRIBUTE_DESIGNATOR [NAME_TAIL]*
function NAME_TAIL return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("NAME_TAIL");
  end if;
  if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    SYMBOL_TABLE.SAVE_CURRENT_ENTRY;
    if (LEFT_PAREN_NAME_TAIL) then
      SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
      return (TRUE);
    else
      SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
      return (FALSE);
    end if;                                 -- if left_paren_name_tail
  elsif (TM.MATCH(TM.TOKEN_PERIOD)) then
    if (P4.SELECTOR) then
      while (NAME_TAIL) loop
        null;
      end loop;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Name tail : expecting selector");
    end if;                                 -- if selector statement
  elsif (TM.MATCH(TM.TOKEN_APOSTROPHE)) then
    SYMBOL_TABLE.SAVE_CURRENT_ENTRY;
```

```
    if (AGGREGATE) then
      while (NAME_TAIL) loop
        null;
      end loop;
      SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
      return (TRUE);
    elsif (ATTRIBUTE_DESIGNATOR) then
      while (NAME_TAIL) loop
        null;
      end loop;
      SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Name tail : expecting aggregate or attribute");
    end if;                                   -- if aggregate statement
  else
    return (FALSE);
  end if;                                     -- if match(token_left_paren)
end NAME_TAIL;

--------------------------------------------------------------------------------

  -- LEFT_PAREN_NAME_TAIL -->  [FORMAL_PARAMETER ?] EXPRESSION [..EXPRESSION ?]
  --                          [, [FORMAL_PARAMETER ?] EXPRESSION [..EXPRESSION ?]]*
  --                          ) [NAME_TAIL]*
function LEFT_PAREN_NAME_TAIL return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("LEFT_PAREN_NAME_TAIL");
  end if;
  if (P4.FORMAL_PARAMETER) then          -- check for optional formal parameter
    null;                                -- before the actual parameter
  end if;                                     -- if formal_parameter statement
  if (EXPRESSION) then
    if (TM.MATCH(TM.TOKEN_RANGE_DOTS)) then
      if not (EXPRESSION) then
        P4.SYNTAX_ERROR("Left paren name tail");
      end if;                                 -- if not expression statement
    end if;                                   -- if match(token_range_dots)
    while (TM.MATCH(TM.TOKEN_COMMA)) loop
      if (P4.FORMAL_PARAMETER) then
        null;
      end if;                                 -- if formal_parameter statement
      if not (EXPRESSION) then
        P4.SYNTAX_ERROR("Left paren name tail");
      end if;                                 -- if not expression statement
      if (TM.MATCH(TM.TOKEN_RANGE_DOTS)) then
        if not (EXPRESSION) then
          P4.SYNTAX_ERROR("Left paren name tail");
        end if;                               -- if not expression statement
      end if;                                 -- if match(token_range_dots)
```

```
    end loop;
    if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
      while (NAME_TAIL) loop
        null;
      end loop;
      return (TRUE);
    else
      return (FALSE);
    end if;                                -- if match(token_right_paren)
  elsif (DISCRETE_RANGE) then
     if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
        while (NAME_TAIL) loop
          null;
        end loop;
        return (TRUE);
      else
       P4.SYNTAX_ERROR("Left paren name tail");
      end if;
   else
      return (FALSE);
    end if;                                -- if match(token_right_paren)
end LEFT_PAREN_NAME_TAIL;

--------------------------------------------------------------------------------

  -- ATTRIBUTE_DESIGNATOR -->  identifier [(EXPRESSION) ?]
  --                      -->  range [(EXPRESSION) ?]
  --                      -->  digits [(EXPRESSION) ?]
  --                      -->  delta [(EXPRESSION) ?]
function ATTRIBUTE_DESIGNATOR return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("ATTRIBUTE_DESIGNATOR");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) or else (TM.MATCH(TM.TOKEN_RANGE)) then
    if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
      if (EXPRESSION) then
        if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
          null;
        else
          P4.SYNTAX_ERROR("Attribute designator");
        end if;                            -- if match(token_right_paren)
      else
        P4.SYNTAX_ERROR("Attribute designator");
      end if;                              -- if expression statement
    end if;                                -- if match(token_left_paren)
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_DIGITS)) or else (TM.MATCH(TM.TOKEN_DELTA)) then
    if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
      if (EXPRESSION) then
        if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
```

```
          null;
        else
          P4.SYNTAX_ERROR("Attribute designator");
        end if;                                 -- if match(token_right_paren)
      else
        P4.SYNTAX_ERROR("Attribute designator");
      end if;                                   -- if expression statement
    end if;                                     -- if match(token_left_paren)
    return (TRUE);
  else
    return (FALSE);
  end if;                                       -- if match(token_identifier)
end ATTRIBUTE_DESIGNATOR;

--------------------------------------------------------------------------------

  -- INTEGER_TYPE_DEFINITION -->  range RANGES
function INTEGER_TYPE_DEFINITION return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("INTEGER_TYPE_DEFINITION");
  end if;
  if (TM.MATCH(TM.TOKEN_RANGE)) then
    if (RANGES) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Integer type definition");
    end if;
  else
    return (FALSE);
  end if;
end INTEGER_TYPE_DEFINITION;

--------------------------------------------------------------------------------

  -- DISCRETE_RANGE -->  RANGES [CONSTRAINT ?]
function DISCRETE_RANGE return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("DISCRETE_RANGE");
  end if;
  if (RANGES) then
    if (CONSTRAINT) then
      null;
    end if;                                     -- if constraint statement
    return (TRUE);
  else
    return (FALSE);
  end if;                                       -- if ranges statement
end DISCRETE_RANGE;
```

```
-------------------------------------------------------------------------------

  -- EXIT_STATEMENT -->  [NAME ?] [when EXPRESSION ?] ;
function EXIT_STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("EXIT_STATEMENT");
  end if;
  if (NAME) then
    null;
  end if;                                  -- if name statement
  if (TM.MATCH(TM.TOKEN_WHEN)) then
    if (EXPRESSION) then
      null;
    else
      P4.SYNTAX_ERROR("Exit statement");
    end if;                                -- if expression statement
  end if;                                  -- if match(token_when)
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE);
  end if;                                  -- if match(token_semicolon)
end EXIT_STATEMENT;

-------------------------------------------------------------------------------

  -- RETURN_STATEMENT -->  [EXPRESSION ?] ;
function RETURN_STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("RETURN_STATEMENT");
  end if;
  if (EXPRESSION) then
    null;
  end if;
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end RETURN_STATEMENT;

-------------------------------------------------------------------------------

  -- GOTO_STATEMENT -->  NAME ;
function GOTO_STATEMENT return boolean is
START_TOKEN  : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
LOCATION_ONE : positive;
use SYMBOL_TABLE;
begin
```

```
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("GOTO_STATEMENT");
  end if;
  if (NAME) then
    TM.MATCHED_TOKEN(START_TOKEN);
    if (SYMBOL_TABLE.FIND_KEY(START_TOKEN.LEXEME(1..START_TOKEN.LEXEME_SIZE))
    = null) then
      SYMBOL_TABLE.INSERT_SYM_TAB(START_TOKEN.LEXEME(1..START_TOKEN.
      LEXEME_SIZE),SYMBOL_TABLE.LABEL_NAME, 0);
    end if;
    LOCATION_ONE := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
    NET_GENERATOR.GO_TO(LOCATION_ONE,
        SYMBOL_TABLE.FIND_KEY(START_TOKEN.LEXEME(1..START_TOKEN.LEXEME_SIZE)));
    CODE_BLOCKER.INCREMENT_STATEMENT_COUNT;
    CODE_BLOCKER.EXIT_CODE_BLOCK(START_TOKEN.SOURCE);
    CODE_BLOCKER.ENTER_CODE_BLOCK(START_TOKEN.SOURCE, "");
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Goto statement");
    end if;                                 -- if match(token_semicolon)
  else
    return (FALSE);
  end if;                                   -- if name statement
end GOTO_STATEMENT;

--------------------------------------------------------------------------------

  -- DELAY_STATEMENT -->  SIMPLE_EXPRESSION ;
function DELAY_STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("DELAY_STATEMENT");
  end if;
  if (SIMPLE_EXPRESSION) then
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Delay statement");
    end if;                                 -- if match(token_semicolon)
  else
    return (FALSE);
  end if;                                   -- if simple_expression statement
end DELAY_STATEMENT;

--------------------------------------------------------------------------------

  -- ABORT_STATEMENT -->  NAME [, NAME]* ;
function ABORT_STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
```

```
    P4.OUT_PUT("ABORT_STATEMENT");
  end if;
  if (NAME) then
    while (TM.MATCH(TM.TOKEN_COMMA)) loop
      if not (NAME) then
        P4.SYNTAX_ERROR("Abort statement");
      end if;                                 -- if not name statement
    end loop;
    if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      P4.SYNTAX_ERROR("Abort statement");
    end  if;                                  -- if match(token_semicolon)
  else
    return (FALSE);
  end if;                                     -- if name statement
end ABORT_STATEMENT;

--------------------------------------------------------------------------------

  -- RAISE_STATEMENT -->  [NAME ?] ;
function RAISE_STATEMENT return boolean is
begin
  if (P4.PRINT_CALLS) then
    P4.OUT_PUT("RAISE_STATEMENT");
  end if;
  if (NAME) then
    null;
  end if;
  if (TM.MATCH(TM.TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end RAISE_STATEMENT;

end PARSER_3;
```

```
--*****************************************************************--
--                                                                 --
--  TITLE:          ADAFLOW                                        --
--                                                                 --
--  MODULE NAME:    PACKAGE PARSER_4                               --
--  FILE NAME:      PARSER4.ADS                                    --
--                                                                 --
--  DATE CREATED:   20 FEB 88                                      --
--  LAST MODIFIED:  28 APR 88                                      --
--                                                                 --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                       --
--                                                                 --
--  BASED ON A MODIFIED ADA GRAMMAR DEVELOPED BY:                  --
--                  LCDR JEFFREY L. NIEDER, USN                    --
--                  LT KARL S. FAIRBANKS, JR., USN                 --
--                  LCDR PAUL M. HERZIG, USN                       --
--                                                                 --
--  DESCRIPTION:  This package defines the functions that          --
--       are the lowest level productions for a top-down,          --
--       recursive descent parser.                                 --
--                                                                 --
--*****************************************************************--
with TEXT_IO, TOKEN_MATCHER;

package PARSER_4 is

  PRINT_CALLS : boolean := FALSE;

  PARSER_ERROR : exception;

  function MULTIPLYING_OPERATOR return boolean;
  function BINARY_ADDING_OPERATOR return boolean;
  function RELATIONAL_OPERATOR return boolean;
  function ENUMERATION_TYPE_DEFINITION return boolean;
  function ENUMERATION_LITERAL return boolean;
  function FORMAL_PARAMETER return boolean;
  function SELECTOR return boolean;

  procedure SYNTAX_ERROR(ERROR_MESSAGE : in string);

  procedure OUT_PUT(FUNCTION_NAME : in string);

end PARSER_4;
```

```
--*****************************************************************--
--                                                                 --
--  TITLE:          ADAFLOW                                        --
--                                                                 --
--  MODULE NAME:    PACKAGE PARSER_4                               --
--  FILE NAME:      PARSER4.ADB                                    --
--                                                                 --
--  DATE CREATEO:   20 FEB 88                                      --
--  LAST MODIFIED:  28 APR 88                                      --
--                                                                 --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                       --
--                                                                 --
--  BASEO ON A MOOIFIEO AOA GRAMMAR OEVELOPEO BY:                  --
--                  LCDR JEFFREY L. NIEDER, USN                    --
--                  LT KARL S. FAIRBANKS, JR., USN                 --
--                  LCOR PAUL M. HERZIG, USN                       --
--                                                                 --
--  OESCRIPTION:  This package implements functions that          --
--       are the lowest level productions for a top-down,          --
--       recursive descent parser.  Each function is preceded      --
--       by the grammar productions they are implementing.         --
--                                                                 --
--*****************************************************************--

with TOKEN_MATCHER, TOKEN_SCANNER, TEXT_IO, SYMBOL_TABLE;

package body PARSER_4 is
  package TM renames TOKEN_MATCHER;

  -- MULTIPLYING_OPERATOR -->  *
  --                      -->  /
  --                      -->  mod
  --                      -->  rem
function MULTIPLYING_OPERATOR return boolean is
begin
  if (PRINT_CALLS) then
    OUT_PUT("MULTIPLYING_OPERATOR");
  end if;
  if (TM.MATCH(TM.TOKEN_ASTERISK)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_SLASH)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_MOD)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_REM)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end MULTIPLYING_OPERATOR;
```

```
--------------------------------------------------------------------------------

  -- BINARY_ADDING_OPERATOR -->  +
  --                        -->  -
  --                        -->  &
function BINARY_ADDING_OPERATOR return boolean is
begin
  if (PRINT_CALLS) then
    OUT_PUT("BINARY_ADDING_OPERATOR");
  end if;
  if (TM.MATCH(TM.TOKEN_PLUS)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_MINUS)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_AMPERSAND)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end BINARY_ADDING_OPERATOR;

--------------------------------------------------------------------------------

  --  RELATIONAL_OPERATOR -->  =
  --                      -->  /=
  --                      -->  <
  --                      -->  <=
  --                      -->  >
  --                      -->  >=
function RELATIONAL_OPERATOR return boolean is
begin
  if (PRINT_CALLS) then
    OUT_PUT("RELATIONAL_OPERATOR");
  end if;
  if (TM.MATCH(TM.TOKEN_EQUALS)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_NOT_EQUALS)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_LESS_THAN)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_LESS_THAN_EQUALS)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_GREATER_THAN)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_GREATER_THAN_EQUALS)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end RELATIONAL_OPERATOR;
```

```
-------------------------------------------------------------------------------

  -- ENUMERATION_TYPE_DEFINITION -->  (ENUMERATION_LITERAL
  --                                   [, ENUMERATION_LITERAL]*)
function ENUMERATION_TYPE_DEFINITION return boolean is
begin
  if (PRINT_CALLS) then
    OUT_PUT("ENUMERATION_TYPE_DEFINITION");
  end if;
  if (TM.MATCH(TM.TOKEN_LEFT_PAREN)) then
    if (ENUMERATION_LITERAL) then
      while (TM.MATCH(TM.TOKEN_COMMA)) loop
        if not (ENUMERATION_LITERAL) then
          SYNTAX_ERROR("Enumeration type definition");
        end if;                           -- if not enumeration_literal
      end loop;
      if (TM.MATCH(TM.TOKEN_RIGHT_PAREN)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Enumeration type definition");
      end if;                             -- if match(token_right_paren)
    else
      SYNTAX_ERROR("Enumeration type definition");
    end if;                               -- if enumeration_literal statement
  else
    return (FALSE);
  end if;                                 -- if match(token_left_paren)
end ENUMERATION_TYPE_DEFINITION;

-------------------------------------------------------------------------------

  -- ENUMERATION_LITERAL -->  identifier
  --                     -->  character_literal
function ENUMERATION_LITERAL return boolean is
begin
  if (PRINT_CALLS) then
    OUT_PUT("ENUMERATION_LITERAL");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_CHARACTER_LITERAL)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end ENUMERATION_LITERAL;

-------------------------------------------------------------------------------

  -- FORMAL_PARAMETER -->  identifier =>
function FORMAL_PARAMETER return boolean is
```

```
PEEK_AHEAD_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
TEST_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
use TOKEN_SCANNER;
begin
  if (PRINT_CALLS) then
    OUT_PUT("FORMAL_PARAMETER");
  end if;
  TEST_TOKEN.LEXEME := (others => ' ');
  TEST_TOKEN.LEXEME(1..2) := "=>";
  TEST_TOKEN.LEXEME_SIZE := 2;
  TEST_TOKEN.TOKEN_TYPE := TOKEN_SCANNER.DELIMITER;
  TM.NEXT_TOKEN(PEEK_AHEAD_TOKEN);
  if (PEEK_AHEAD_TOKEN = TEST_TOKEN) then
    if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
      if (TM.MATCH(TM.TOKEN_ARROW)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Formal parameter");
      end if;                              -- if match(token_arrow)
    else
      SYNTAX_ERROR("Formal parameter");
    end if;                                -- if match(token_identifier)
  else
    return (FALSE);
  end if;
end FORMAL_PARAMETER;

--------------------------------------------------------------------------------

  -- SELECTOR -->  identifier
  --          -->  character_literal
  --          -->  string_literal
  --          -->  all
function SELECTOR return boolean is
SEARCH_POINTER : SYMBOL_TABLE.SYM_TAB_ACCESS;
SEARCH_TOKEN   : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
use SYMBOL_TABLE;
begin
  if (PRINT_CALLS) then
    OUT_PUT("SELECTOR");
  end if;
  if (TM.MATCH(TM.TOKEN_IDENTIFIER)) then
    TM.MATCHED_TOKEN(SEARCH_TOKEN);
    SEARCH_POINTER := SYMBOL_TABLE.RETRIEVE_SYM;
    if (SEARCH_POINTER /= null) then
      SEARCH_POINTER := SYMBOL_TABLE.SELECT_COMPONENT(SEARCH_TOKEN.
                                  LEXEME(1..SEARCH_TOKEN.LEXEME_SIZE));
    end if;
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_CHARACTER_LITERAL)) then
    return (TRUE);
```

```
  elsif (TM.MATCH(TM.TOKEN_STRING_LITERAL)) then
    return (TRUE);
  elsif (TM.MATCH(TM.TOKEN_ALL)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end SELECTOR;

procedure SYNTAX_ERROR(ERROR_MESSAGE : in string) is
begin
  TEXT_IO.new_line(2);
  TEXT_IO.put("Incomplete ");
  TEXT_IO.put(ERROR_MESSAGE);
  TEXT_IO.put(" at line number ");
  TEXT_IO.put(positive'IMAGE(TM.LINES_CHECKED));
  TEXT_IO.new_line(2);
  raise PARSER_ERROR;
end SYNTAX_ERROR;

procedure OUT_PUT(FUNCTION_NAME : in string) is
TOP_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
use TEXT_IO, TOKEN_SCANNER;
begin
  TOKEN_MATCHER.CURRENT_TOKEN(TOP_TOKEN);
  put(FUNCTION_NAME); set_col(40);
  if (TOP_TOKEN.TOKEN_TYPE /= TOKEN_SCANNER.EOF) then
    for LEXEME_INDEX in 1..TOP_TOKEN.LEXEME_SIZE loop
      put(TOP_TOKEN.LEXEME(LEXEME_INDEX));
    end loop;
  end if;
  new_line; set_col(40);
  put_line(TOKEN_SCANNER.TOKEN_CLASS'IMAGE(TOP_TOKEN.TOKEN_TYPE));
end OUT_PUT;

end PARSER_4;
```

APPENDIX D

"ADAFLOW" PROGRAM LISTING - NET GENERATOR

```
--***************************************************************--
--                                                               --
--  TITLE:          ADAFLOW                                      --
--                                                               --
--  MODULE NAME:    PACKAGE NET_GENERATOR                        --
--  FILE NAME:      NET.ADS                                      --
--                                                               --
--  DATE CREATED:   12 MAR 88                                    --
--  LAST MODIFIED:  28 APR 88                                    --
--                                                               --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                     --
--                                                               --
--  DESCRIPTION:  This package contains the procedures which     --
--                define the interface to the net generator.     --
--                                                               --
--***************************************************************--

with SYMBOL_TABLE;

package NET_GENERATOR is

  NET_GENERATOR_OVERFLOW : exception;

  procedure START(RUN_UNIT_NAME : in SYMBOL_TABLE.SYM_TAB_ACCESS);
  -- post - Defines a either a subprogram place or task place that has
  --        an initial marking in the petri net model.

  procedure DECISION_START(START_PLACE : in positive;
                           END_PLACE   : in SYMBOL_TABLE.SYM_TAB_ACCESS);
  -- post - Defines a place that is the root place of a multi-way decision
  --        path and it's corresponding end label.

  procedure DECISION_OR(END_PATH_PLACE : in positive);
  -- post - Ends the current path of a multi-way decision and starts the
  --        next path.  The decision start place is reactivated as the
  --        current block number.

  procedure EXPLICIT_DECISION_OR;
  -- post - Ends the current path of a multi-way decision and starts the
  --        next path.  The decision start place is reactivated as the
  --        current block number.
```

```
procedure END_DECISION(END_PATH_PLACE     : in positive);
-- post - Ends the current path of a multi-way decision and terminates
--        the multi-way decision.

procedure EXPLICIT_END_DECISION;
-- post - Ends the current path of a multi-way decision and terminates
--        the multi-way decision.

procedure CALL(CURRENT_LOCATION   : in positive;
               PROCEDURE_LOCATION : in SYMBOL_TABLE.SYM_TAB_ACCESS);
-- pre  - The procedure location must be the current entry in the
--        symbol table.
-- post - The abstract grammar for a procedure call is generated.

procedure ENTRY_CALL(CURRENT_LOCATION : in positive;
                     ENTRY_LOCATION   : in SYMBOL_TABLE.SYM_TAB_ACCESS);
-- pre  - The entry location must be the current entry in the
--        symbol table.
-- post - The abstract grammar for a task entry is generated.

procedure TASK_ACCEPT(CURRENT_LOCATION : in positive;
                      ENTRY_LOCATION   : in positive);
-- post - The abstract grammar for a task accept is generated.

procedure END_ACCEPT(CURRENT_LOCATION : in positive;
                     ENTRY_END        : in positive);
-- post - The abstract grammar for the end of an accept statement is
--        generated.

procedure EXPLICIT_END_ACCEPT(ENTRY_END : in positive);
-- post - The abstract grammar for the end of an accept statement is
--        generated.

procedure GO_TO(CURRENT_LOCATION : in positive;
                GO_TO_LOCATION   : in SYMBOL_TABLE.SYM_TAB_ACCESS);
-- post - The abstract grammar for a goto statement is generated.

procedure END_LOOP(END_LOCATION : in positive;
                   LOOP_START   : in SYMBOL_TABLE.SYM_TAB_ACCESS);
-- post - The abstract grammar for a loop is generated.

procedure CONNECT_BLOCKS(CURRENT_LOCATION : in positive;
                         NEXT_LOCATION    : in positive);
-- post - used to explicitly declare a transition between two known
--        code blocks.  The abstract grammar for a transition between
--        two petri net places is generated.

procedure EXPLICIT_END(NEXT_LOCATION : in positive);
-- post - The current forest is terminated and a new forest is begun.
```

```
  procedure TRANSLATE_TO_PEANUT;
  -- post - used to translate the abstract petri net grammar to a
  --        text file used as an input file to P-NUT petri net analyzer.
  --        Produces two files: 1)  a.out -     P-NUT input file
  --                            2)  place.dat - text file that describes all
  --                                            the places that exist in the
  --                                            petri net and/or the
  --                                            places relation to the
  --                                            original source code.
  --        The net generator and code blocker are reset to their
  --        initial states.

  procedure RESET_NET_GENERATOR;
  -- post - The net generator is returned to it's initial state.

end NET_GENERATOR;
```

```
--*************************************************************--
--                                                             --
--  TITLE:          ADAFLOW                                    --
--                                                             --
--  MODULE NAME:    PACKAGE NET_GENERATOR                      --
--  FILE NAME:      NET.ADB                                    --
--                                                             --
--  DATE CREATED:   12 MAR 88                                  --
--  LAST MODIFIED:  28 APR 88                                  --
--                                                             --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                   --
--                                                             --
--  DESCRIPTION:  This package contains the procedures which   --
--                implement the interface to the net generator. --
--                                                             --
--*************************************************************--

with TOKEN_SCANNER,
     GENERIC_LIST,
     GENERIC_STACK,
     UNCHECKED_DEALLOCATION,
     SYMBOL_TABLE,
     CODE_BLOCKER,
     TEXT_IO,
     IO_EXCEPTIONS;

package body NET_GENERATOR is

  DUMMY_SOURCE : TOKEN_SCANNER.SOURCE_RECORD;

  type PETRI_IDENTIFIER_TYPE is (PLACE, TRANSITION);

  type LIST_NODE is
    record
      PETRI_TAG : PETRI_IDENTIFIER_TYPE;
      SYMBOL    : SYMBOL_TABLE.SYM_TAB_ACCESS := null;
    end record;

  type LIST_NODE_POINTER is access LIST_NODE;

  package NEST_STACK is new GENERIC_STACK(LIST_NODE_POINTER);
  NS : NEST_STACK.STACK;

  TRANSITION_POINTER : LIST_NODE_POINTER;
  DECISION_ROOT      : LIST_NODE_POINTER := null;
  DECISION_TAIL      : LIST_NODE_POINTER := null;

  package ABSTRACT_SYNTAX_LIST is

    type LIST_INSTANCE is private;
    type LIST is access LIST_INSTANCE;
```

```
    LIST_OVERFLOW  : exception;
    LIST_UNDERFLOW : exception;

-- Operations:  If the list is not empty, then one of the nodes is designated
--    as the current node.  Ocaasionally, in the postcondition, it is necessary
--    to refer to the list of the current node as they were immediately before
--    execution of the operation.  L-pre and c-pre, respectively, are employed
--    for these references.

    procedure FIND_FIRST(L : in out LIST);
    -- pre  - The list L is not empty.
    -- post - The first node is the current node.
    -- exceptions raised - LIST_UNDERFLOW if L is empty.

    procedure FIND_NEXT(L : in out LIST);
    -- pre  - The list L is not empty and the last node is not the current node.
    -- post - c-next in L is the current node.
    -- exceptions raised - LIST_UNDERFLOW if L is empty.
    --                   - LIST_OVERFLOW  if the last node is the current node.

    procedure FIND_PREVIOUS(L : in out LIST);
    -- pre  - The list L is not empty and the first node is not the current node.
    -- post - c-prior in L is the current node.
    -- exceptions raised - LIST_UNDERFLOW if L is empty or c is the first node.

    procedure FIND_LAST(L : in out LIST);
    -- pre  - The list L is not empty.
    -- post - The last node in L is the current node.
    -- exceptions raised - LIST_UNDERFLOW if L is empty.

    procedure RETRIEVE(L : in LIST; ITEM : out LIST_NODE_POINTER);
    -- pre  - The list L is not empty.
    -- post - ITEM contains the value of the element in the current node.
    -- exceptions raised - LIST_UNDERFLOW if L is empty.

    procedure UPDATE(L : in out LIST; ITEM : in LIST_NODE_POINTER);
    -- pre  - The list L is not empty.
    -- post - The current node in L contains ITEM as its element.
    -- exceptions raised - LIST_UNDERFLOW if L is empty.

    procedure INSERT(L : in out LIST; ITEM : in LIST_NODE_POINTER);
    -- pre  - The number of nodes in L has not reached its bound.
    -- post - A node containing ITEM is the last node in the list, and the last
    --        node in L-pre, if any, is its predecessor.  The node containing
    --        ITEM is the current node.
    -- exceptions raised - LIST_OVERFLOW if L has reached its bound.

    procedure DELETE(L : in out LIST);
    -- pre  - The list L is not empty.
    -- post - c-pre in not in the list L.  If c-pre was the first node,
```

```
  --        then c-next, if it exists, is the successor of c-prior.  If the
  --        list L is not empty, then the last node is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.

  function  SIZE_OF(L : in LIST) return natural;
  -- post - SIZE_OF is the number of nodes in list L.

  function  EMPTY(L : in LIST) return boolean;
  -- post - If the list L has no nodes then EMPTY is true, else EMPTY is
  --        false.

  function  FULL(L : in LIST) return boolean;
  -- post - If the number of nodes in the list L has reached the maximum
  --        allowed, then FULL is true, else FULL is false.

  function  FIRST(L : in LIST) return boolean;
  -- pre  - The list L is not empty.
  -- post - If the first node is the current node in L then FIRST is true, else
  --        FIRST is false.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.

  function  LAST(L : in LIST) return boolean;
  -- pre  - The list L is not empty.
  -- post - If the last node is the current node in L then LAST is true, else
  --        LAST is false.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.

  procedure CREATE(L : in out LIST; SUCCESS :  out boolean);
  -- post - If a list L can be created then L exists and is empty, and SUCCESS
  --        is TRUE else SUCCESS is FALSE.

  procedure DISPOSE(L : in out LIST);
  -- post - L-pre does not exist.

private

  type NODE;
  type NODE_POINTER is access NODE;
  type NODE is
    record
      ELEMENT : LIST_NODE_POINTER;
      NEXT    : NODE_POINTER;
    end record;
  type LIST_INSTANCE is
    record
      HEAD    : NODE_POINTER := null;
      TAIL    : NODE_POINTER := null;
      CURRENT : NODE_POINTER := null;
      SIZE    : natural := 0;
    end record;
```

```
end ABSTRACT_SYNTAX_LIST;

package FOREST_LIST is new GENERIC_LIST(ABSTRACT_SYNTAX_LIST.LIST);

FOREST : FOREST_LIST.LIST;

START_SYNTAX : ABSTRACT_SYNTAX_LIST.LIST;
STOP_PLACES  : ABSTRACT_SYNTAX_LIST.LIST;

package body ABSTRACT_SYNTAX_LIST is

  procedure FREE_NODE is new UNCHECKED_DEALLOCATION(NODE, NODE_POINTER);
  procedure FREE_LIST is new UNCHECKED_DEALLOCATION(LIST_INSTANCE, LIST);
  procedure FREE_SYM_REC is new UNCHECKED_DEALLOCATION(SYMBOL_TABLE.
                                                       SYM_TAB_RECORD,
                                                       SYMBOL_TABLE.
                                                       SYM_TAB_ACCESS);

  procedure FIND_FIRST(L : in out LIST) is
  -- pre  - The list L is not empty.
  -- post - The first node is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    L.CURRENT := L.HEAD;
  end FIND_FIRST;

  procedure FIND_NEXT(L : in out LIST) is
  -- pre  - The list L is not empty and the last node is not the current node.
  -- post - c-next in L is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  --                   - LIST_OVERFLOW  if the last node is the current node.
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    if (LAST(L)) then
      raise LIST_OVERFLOW;
    end if;
    L.CURRENT := L.CURRENT.NEXT;
  end FIND_NEXT;

  procedure FIND_PREVIOUS(L : in out LIST) is
  -- pre  - The list L is not empty and the first node is not the current node.
  -- post - c-prior in L is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty or c is the first node.
  TEMP_POINTER : NODE_POINTER;
  begin
    if (EMPTY(L) or FIRST(L)) then
```

```
    raise LIST_UNDERFLOW;
  end if;
  TEMP_POINTER := L.HEAD;
  while (TEMP_POINTER.NEXT /= L.CURRENT) loop
    TEMP_POINTER := TEMP_POINTER.NEXT;
  end loop;
  L.CURRENT := TEMP_POINTER;
end FIND_PREVIOUS;

procedure FIND_LAST(L : in out LIST) is
-- pre  - The list L is not empty.
-- post - The last node in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  while (not LAST(L)) loop
    FIND_NEXT(L);
  end loop;
end FIND_LAST;

procedure RETRIEVE(L : in LIST; ITEM : out LIST_NODE_POINTER) is
-- pre  - The list L is not empty.
-- post - ITEM contains the value of the element in the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  ITEM := L.CURRENT.ELEMENT;
end RETRIEVE;

procedure UPDATE(L : in out LIST; ITEM : in LIST_NODE_POINTER) is
-- pre  - The list L is not empty.
-- post - The current node in L contains ITEM as its element.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  L.CURRENT.ELEMENT := ITEM;
end UPDATE;

procedure INSERT(L : in out LIST; ITEM : in LIST_NODE_POINTER) is
-- pre  - The number of nodes in L has not reached its bound.
-- post - A node containing ITEM is the last node in the list, and the last
--        node in L-pre, if any, is its predecessor.  The node containing
--        ITEM is the current node.
-- exceptions raised - LIST_OVERFLOW if L has reached its bound.
TEMP_POINTER : NODE_POINTER;
```

```
use SYMBOL_TABLE;
begin
  if (FULL(L)) then
    raise LIST_OVERFLOW;
  end if;
  TEMP_POINTER := new NODE'(ITEM, null);
  TEMP_POINTER.ELEMENT.SYMBOL.REFERENCE_COUNT :=
    natural'SUCC(TEMP_POINTER.ELEMENT.SYMBOL.REFERENCE_COUNT);
  if (L.HEAD = null) then
    L.HEAD := TEMP_POINTER;
    L.TAIL := TEMP_POINTER;
  else
    L.TAIL.NEXT := TEMP_POINTER;
    L.TAIL      := TEMP_POINTER;
  end if;
  L.CURRENT := TEMP_POINTER;
  L.SIZE := L.SIZE + 1;
end INSERT;

procedure DELETE(L : in out LIST) is
-- pre  - The list L is not empty.
-- post - c-pre in not in the list L.  If c-pre was the first node,
--        then c-next, if it exists, is the successor of c-prior.  If the
--        list L is not empty, then the last node is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
TEMP_POINTER : NODE_POINTER;
use SYMBOL_TABLE;
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  if (L.CURRENT /= L.HEAD) then
    TEMP_POINTER := L.HEAD;
    while (TEMP_POINTER.NEXT /= L.CURRENT) loop
      TEMP_POINTER := TEMP_POINTER.NEXT;
    end loop;
    TEMP_POINTER.NEXT := L.CURRENT.NEXT;
    if (L.CURRENT = L.TAIL) then
      L.TAIL := TEMP_POINTER;
    end if;
  else
    if (L.HEAD = L.TAIL) then
      L.TAIL := null;
    end if;
    L.HEAD := L.HEAD.NEXT;
  end if;
  if (L.CURRENT.ELEMENT.SYMBOL.REFERENCE_COUNT > 1) then
    L.CURRENT.ELEMENT.SYMBOL.REFERENCE_COUNT :=
      positive'PRED(L.CURRENT.ELEMENT.SYMBOL.REFERENCE_COUNT);
  else
    FREE_SYM_REC(L.CURRENT.ELEMENT.SYMBOL);
```

```
  end if;
  FREE_NODE(L.CURRENT);
  L.CURRENT := L.TAIL;
  L.SIZE := L.SIZE - 1;
end DELETE;

function  SIZE_OF(L : in LIST) return natural is
-- post - SIZE_OF is the number of nodes in list L.
begin
  return (L.SIZE);
end SIZE_OF;

function  EMPTY(L : in LIST) return boolean is
-- post - If the list L has no nodes then EMPTY is true, else EMPTY is
--         false.
begin
  return (L.HEAD = null);
end EMPTY;

function  FULL(L : in LIST) return boolean is
-- post - If the number of nodes in the list L has reached the maximum
--         allowed, then FULL is true, else FULL is false.
TEMP_POINTER : NODE_POINTER;
begin
  TEMP_POINTER := new NODE;
  FREE_NODE(TEMP_POINTER);
  return (FALSE);
exception
  when STORAGE_ERROR =>
    return (TRUE);
  when others =>
    raise;
end FULL;

function  FIRST(L : in LIST) return boolean is
-- pre  - The list L is not empty.
-- post - If the first node is the current node in L then FIRST is true, else
--         FIRST is false.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  return (L.CURRENT = L.HEAD);
end FIRST;

function  LAST(L : in LIST) return boolean is
-- pre  - The list L is not empty.
-- post - If the last node is the current node in L then LAST is true, else
--         LAST is false.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
```

```
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    return (L.CURRENT = L.TAIL);
  end LAST;

  procedure CREATE(L : in out LIST; SUCCESS :  out boolean) is
  -- post - If a list L can be created then L exists and is empty, and SUCCESS
  --        is TRUE else SUCCESS is FALSE.
  begin
    L := new LIST_INSTANCE'(null, null, null, 0);
    SUCCESS := TRUE;
  exception
    when STORAGE_ERROR =>
      SUCCESS := FALSE;
    when others =>
      raise;
  end CREATE;

  procedure DISPOSE(L : in out LIST) is
  -- post - L-pre does not exist.
  begin
    if (not EMPTY(L)) then
      FIND_LAST(L);
      while (not EMPTY(L)) loop
        DELETE(L);
      end loop;
    end if;
    FREE_LIST(L);
  end DISPOSE;

end ABSTRACT_SYNTAX_LIST;

function CREATE_DUMMY_PLACE(LABEL : in string)
                                        return LIST_NODE_POINTER is
-- post - a place is created with a unique code block number and given
--        a tag denoted by LABEL.  CREATE_DUMMY_PLACE returns a pointer
--        to a syntax list node that now contains this place.
LOCATION : positive;
TEMP_POINTER : LIST_NODE_POINTER;
begin
  CODE_BLOCKER.ENTER_CODE_BLOCK(DUMMY_SOURCE, LABEL);
  LOCATION := CODE_BLOCKER.CURRENT_CODE_BLOCK_NUMBER;
  CODE_BLOCKER.EXIT_CODE_BLOCK(DUMMY_SOURCE);
  TEMP_POINTER := new LIST_NODE;
  TEMP_POINTER.PETRI_TAG := PLACE;
  TEMP_POINTER.SYMBOL := new SYMBOL_TABLE.SYM_TAB_RECORD;
  TEMP_POINTER.SYMBOL.NAME := (others => ' ');
  TEMP_POINTER.SYMBOL.NAME_LENGTH := 0;
  TEMP_POINTER.SYMBOL.LOCATION := LOCATION;
```

```
  TEMP_POINTER.SYMBOL.REFERENCE_COUNT := 0;
  return (TEMP_POINTER);
exception
  when STORAGE_ERROR =>
    raise NET_GENERATOR_OVERFLOW;
  when others =>
    raise;
end CREATE_DUMMY_PLACE;

function NUMBER_TO_LIST_NODE(CURRENT_LOCATION : in positive)
                                                return LIST_NODE_POINTER is
-- post - NUMBER_TO_LIST_NODE returns a pointer
--        to a syntax list node that now contains this place.
TEMP_POINTER : LIST_NODE_POINTER;
begin
  TEMP_POINTER := new LIST_NODE;
  TEMP_POINTER.PETRI_TAG := PLACE;
  TEMP_POINTER.SYMBOL := new SYMBOL_TABLE.SYM_TAB_RECORD;
  TEMP_POINTER.SYMBOL.NAME := (others => ' ');
  TEMP_POINTER.SYMBOL.NAME_LENGTH := 0;
  TEMP_POINTER.SYMBOL.LOCATION := CURRENT_LOCATION;
  TEMP_POINTER.SYMBOL.REFERENCE_COUNT := 0;
  return (TEMP_POINTER);
exception
  when STORAGE_ERROR =>
    raise NET_GENERATOR_OVERFLOW;
  when others =>
    raise;
end NUMBER_TO_LIST_NODE;

function POINTER_TO_LIST_NODE(LOCATION : in SYMBOL_TABLE.SYM_TAB_ACCESS)
                                                return LIST_NODE_POINTER is
-- post - POINTER_TO_LIST_NODE returns a pointer
--        to a syntax list node that now contains this place.
TEMP_POINTER : LIST_NODE_POINTER;
begin
  TEMP_POINTER := new LIST_NODE;
  TEMP_POINTER.PETRI_TAG := PLACE;
  TEMP_POINTER.SYMBOL := LOCATION;
  return (TEMP_POINTER);
exception
  when STORAGE_ERROR =>
    raise NET_GENERATOR_OVERFLOW;
  when others =>
    raise;
end POINTER_TO_LIST_NODE;

procedure NEW_SYNTAX_LIST is
-- pre  - The forest size has not reached it's bound.
-- post - An empty syntax list is inserted into the forest and becomes the
 -        current element in the forest.
```

```
TEMP_SYNTAX : ABSTRACT_SYNTAX_LIST.LIST;
SUCCESS     : boolean;
begin
  ABSTRACT_SYNTAX_LIST.CREATE(TEMP_SYNTAX, SUCCESS);
  if (not SUCCESS) then
    raise NET_GENERATOR_OVERFLOW;
  end if;
  if (not FOREST_LIST.FULL(FOREST)) then
    FOREST_LIST.INSERT(FOREST, TEMP_SYNTAX);
  else
    raise NET_GENERATOR_OVERFLOW;
  end if;
end NEW_SYNTAX_LIST;

procedure INITIALIZE_NET_GENERATOR is
SUCCESS      : boolean;
begin
  DUMMY_SOURCE.FILE_NAME      := (others => ' ');
  DUMMY_SOURCE.FILE_NAME_SIZE := 0;
  DUMMY_SOURCE.LINE_NUMBER    := 0;
  ABSTRACT_SYNTAX_LIST.CREATE(START_SYNTAX, SUCCESS);
  if (not SUCCESS) then
    raise NET_GENERATOR_OVERFLOW;
  end if;
  ABSTRACT_SYNTAX_LIST.INSERT(START_SYNTAX,
                              CREATE_DUMMY_PLACE("START"));
  TRANSITION_POINTER := new LIST_NODE;
  TRANSITION_POINTER.PETRI_TAG := TRANSITION;
  TRANSITION_POINTER.SYMBOL := new SYMBOL_TABLE.SYM_TAB_RECORD;
  TRANSITION_POINTER.SYMBOL.NAME := (others => ' ');
  TRANSITION_POINTER.SYMBOL.NAME_LENGTH := 0;
  TRANSITION_POINTER.SYMBOL.LOCATION := 0;
  TRANSITION_POINTER.SYMBOL.REFERENCE_COUNT := 0;
  ABSTRACT_SYNTAX_LIST.INSERT(START_SYNTAX, TRANSITION_POINTER);
  ABSTRACT_SYNTAX_LIST.CREATE(STOP_PLACES, SUCCESS);
  if (not SUCCESS) then
    raise NET_GENERATOR_OVERFLOW;
  end if;
  FOREST_LIST.CREATE(FOREST, SUCCESS);
  if (not SUCCESS) then
    raise NET_GENERATOR_OVERFLOW;
  end if;
  NEST_STACK.CREATE(NS, SUCCESS);
  if (not SUCCESS) then
    raise NET_GENERATOR_OVERFLOW;
  end if;
  NEW_SYNTAX_LIST;
exception
  when STORAGE_ERROR =>
    raise NET_GENERATOR_OVERFLOW;
  when others =>
```

```
    raise;
end INITIALIZE_NET_GENERATOR;

procedure RESET_NET_GENERATOR is
-- post - The net generator is returned to it's initial state.
TEMP_ASL : ABSTRACT_SYNTAX_LIST.LIST;
SUCCESS  : boolean;
begin
  ABSTRACT_SYNTAX_LIST.DISPOSE(START_SYNTAX);
  if (not FOREST_LIST.EMPTY(FOREST)) then
    FOREST_LIST.FIND_LAST(FOREST);
    while (not FOREST_LIST.EMPTY(FOREST)) loop
      FOREST_LIST.RETRIEVE(FOREST, TEMP_ASL);
      ABSTRACT_SYNTAX_LIST.DISPOSE(TEMP_ASL);
      FOREST_LIST.DELETE(FOREST);
    end loop;
  end if;
  ABSTRACT_SYNTAX_LIST.DISPOSE(STOP_PLACES);
  ABSTRACT_SYNTAX_LIST.CREATE(START_SYNTAX, SUCCESS);
  if (not SUCCESS) then
    raise NET_GENERATOR_OVERFLOW;
  end if;
  ABSTRACT_SYNTAX_LIST.CREATE(STOP_PLACES, SUCCESS);
  if (not SUCCESS) then
    raise NET_GENERATOR_OVERFLOW;
  end if;
  ABSTRACT_SYNTAX_LIST.INSERT(START_SYNTAX,
                              CREATE_DUMMY_PLACE("START"));
  TRANSITION_POINTER := new LIST_NODE;
  TRANSITION_POINTER.PETRI_TAG := TRANSITION;
  TRANSITION_POINTER.SYMBOL := new SYMBOL_TABLE.SYM_TAB_RECORD;
  TRANSITION_POINTER.SYMBOL.NAME := (others => ' ');
  TRANSITION_POINTER.SYMBOL.NAME_LENGTH := 0;
  TRANSITION_POINTER.SYMBOL.LOCATION := 0;
  TRANSITION_POINTER.SYMBOL.REFERENCE_COUNT := 0;
  ABSTRACT_SYNTAX_LIST.INSERT(START_SYNTAX, TRANSITION_POINTER);
  NEW_SYNTAX_LIST;
end RESET_NET_GENERATOR;

function  IS_COMPLETE return boolean is
-- post - If the current syntax list in the forest is empty, then
--        IS_COMPLETE returns true, else IS_COMPLETE returns false.
TEMP_SYNTAX : ABSTRACT_SYNTAX_LIST.LIST;
begin
  FOREST_LIST.RETRIEVE(FOREST, TEMP_SYNTAX);
  return (ABSTRACT_SYNTAX_LIST.EMPTY(TEMP_SYNTAX));
end IS_COMPLETE;

procedure INSERT_FOREST(TRANS_OR_PLACE : in LIST_NODE_POINTER) is
-- post - The specified transition or place is inserted into the forest
--        in the current syntax list.
```

```
TEMP_LIST : ABSTRACT_SYNTAX_LIST.LIST;
begin
  FOREST_LIST.RETRIEVE(FOREST, TEMP_LIST);
  ABSTRACT_SYNTAX_LIST.INSERT(TEMP_LIST, TRANS_OR_PLACE);
  FOREST_LIST.UPOATE(FOREST, TEMP_LIST);
end INSERT_FOREST;

procedure START(RUN_UNIT_NAME : in SYMBOL_TABLE.SYM_TAB_ACCESS) is
-- post - Oefines a either a subprogram place or task place that has
--        an initial marking in the petri net model.
RUN_UNIT_NOOE : LIST_NODE_POINTER;
ENO_MARKER    : SYMBOL_TABLE.SYM_TAB_ACCESS;
begin
  RUN_UNIT_NOOE := POINTER_TO_LIST_NOOE(RUN_UNIT_NAME);
  ABSTRACT_SYNTAX_LIST.INSERT(START_SYNTAX, RUN_UNIT_NOOE);
  SYMBOL_TABLE.SAVE_CURRENT_ENTRY;
  END_MARKER := SYMBOL_TABLE.FINO_KEY(RUN_UNIT_NAME.NAME(1..
                                RUN_UNIT_NAME.NAME_LENGTH));

  ENO_MARKER := SYMBOL_TABLE.SELECT_COMPONENT("ENO");
  ABSTRACT_SYNTAX_LIST.INSERT(STOP_PLACES,
                              POINTER_TO_LIST_NOOE(ENO_MARKER));

  SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
end START;

procedure DECISION_START(START_PLACE : in positive;
                         ENO_PLACE   : in SYMBOL_TABLE.SYM_TAB_ACCESS) is

-- post - Defines a place that is the root place of a multi-way decision
--        path and it's corresponding end label.

begin
  NEST_STACK.PUSH(NS, DECISION_ROOT);
  NEST_STACK.PUSH(NS, OECISION_TAIL);
  DECISION_ROOT := NUMBER_TO_LIST_NOOE(START_PLACE);
  DECISION_TAIL := POINTER_TO_LIST_NOOE(END_PLACE);
end OECISION_START;

procedure DECISION_OR(END_PATH_PLACE : in positive) is
-- post - Ends the current path of a multi-way decision and starts the
--        next path.  The decision start place is reactivated as the
--        current block number.
START_NOOE : LIST_NOOE_POINTER;
begin
  START_NODE := NUMBER_TO_LIST_NODE(END_PATH_PLACE);
  if (not IS_COMPLETE) then
    INSERT_FOREST(START_NODE);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(START_NODE);
  INSERT_FOREST(TRANSITION_POINTER);
  INSERT_FOREST(DECISION_TAIL);
  NEW_SYNTAX_LIST;
```

```
  CODE_BLOCKER.REACTIVATE_CODE_BLOCK(DECISION_ROOT.SYMBOL.LOCATION);
end DECISION_OR;

procedure EXPLICIT_DECISION_OR is
-- post - Ends the current path of a multi-way decision and starts the
--        next path.  The decision start place is reactivated as the
--        current block number.
begin
  if (not IS_COMPLETE) then
    INSERT_FOREST(DECISION_TAIL);
    NEW_SYNTAX_LIST;
    CODE_BLOCKER.REACTIVATE_CODE_BLOCK(DECISION_ROOT.SYMBOL.LOCATION);
  end if;
end EXPLICIT_DECISION_OR;

procedure END_DECISION(END_PATH_PLACE : in positive) is
-- post - Ends the current path of a multi-way decision and terminates
--        the multi-way decision.
START_NODE : LIST_NODE_POINTER;
begin
  START_NODE := NUMBER_TO_LIST_NODE(END_PATH_PLACE);
  if (not IS_COMPLETE) then
    INSERT_FOREST(START_NODE);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(START_NODE);
  INSERT_FOREST(TRANSITION_POINTER);
  INSERT_FOREST(DECISION_TAIL);
  NEW_SYNTAX_LIST;
  INSERT_FOREST(DECISION_TAIL);
  INSERT_FOREST(TRANSITION_POINTER);
  NEST_STACK.POP(NS, DECISION_TAIL);
  NEST_STACK.POP(NS, DECISION_ROOT);
end END_DECISION;

procedure EXPLICIT_END_DECISION is
-- post - Ends the current path of a multi-way decision and terminates
--        the multi-way decision.
begin
  if (not IS_COMPLETE) then
    INSERT_FOREST(DECISION_TAIL);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(DECISION_TAIL);
  INSERT_FOREST(TRANSITION_POINTER);
  NEST_STACK.POP(NS, DECISION_TAIL);
  NEST_STACK.POP(NS, DECISION_ROOT);
end EXPLICIT_END_DECISION;

procedure CALL(CURRENT_LOCATION   : in positive;
               PROCEDURE_LOCATION : in SYMBOL_TABLE.SYM_TAB_ACCESS) is
```

```
-- pre  - The procedure location must be the current entry in the
--        symbol table.
-- post - The abstract grammar for a procedure call is generated.
START_NODE : LIST_NODE_POINTER;
WAIT_NODE  : LIST_NODE_POINTER;
TEMP_POINTER : SYMBOL_TABLE.SYM_TAB_ACCESS;
begin
  START_NODE := NUMBER_TO_LIST_NODE(CURRENT_LOCATION);
  WAIT_NODE  := CREATE_DUMMY_PLACE("WAIT RETURN");
  SYMBOL_TABLE.SAVE_CURRENT_ENTRY;
  TEMP_POINTER := SYMBOL_TABLE.SELECT_COMPONENT("END");
  SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
  if (not IS_COMPLETE) then
    INSERT_FOREST(START_NODE);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(START_NODE);
  INSERT_FOREST(TRANSITION_POINTER);
  INSERT_FOREST(POINTER_TO_LIST_NODE(PROCEDURE_LOCATION));
  INSERT_FOREST(WAIT_NODE);
  NEW_SYNTAX_LIST;
  INSERT_FOREST(WAIT_NODE);
  INSERT_FOREST(POINTER_TO_LIST_NODE(TEMP_POINTER));
  INSERT_FOREST(TRANSITION_POINTER);
end CALL;

procedure ENTRY_CALL(CURRENT_LOCATION : in positive;
                     ENTRY_LOCATION   : in SYMBOL_TABLE.SYM_TAB_ACCESS) is
-- pre  - The entry location must be the current entry in the
--        symbol table.
-- post - The abstract grammar for a task entry is generated.
START_NODE : LIST_NODE_POINTER;
WAIT_NODE  : LIST_NODE_POINTER;
TEMP_POINTER : SYMBOL_TABLE.SYM_TAB_ACCESS;
begin
  START_NODE := NUMBER_TO_LIST_NODE(CURRENT_LOCATION);
  WAIT_NODE  := CREATE_DUMMY_PLACE("WAIT RENDEZVOUS");
  SYMBOL_TABLE.SAVE_CURRENT_ENTRY;
  TEMP_POINTER := SYMBOL_TABLE.SELECT_COMPONENT("END");
  SYMBOL_TABLE.RESTORE_CURRENT_ENTRY;
  if (not IS_COMPLETE) then
    INSERT_FOREST(START_NODE);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(START_NODE);
  INSERT_FOREST(TRANSITION_POINTER);
  INSERT_FOREST(POINTER_TO_LIST_NODE(ENTRY_LOCATION));
  INSERT_FOREST(WAIT_NODE);
  NEW_SYNTAX_LIST;
  INSERT_FOREST(WAIT_NODE);
  INSERT_FOREST(POINTER_TO_LIST_NODE(TEMP_POINTER));
```

```
  INSERT_FOREST(TRANSITION_POINTER);
end ENTRY_CALL;

procedure TASK_ACCEPT(CURRENT_LOCATION : in positive;
                      ENTRY_LOCATION   : in positive) is
-- post - The abstract grammar for a task accept is generated.
START_NODE : LIST_NODE_POINTER;
begin
  START_NODE := NUMBER_TO_LIST_NODE(CURRENT_LOCATION);
  if (not IS_COMPLETE) then
    INSERT_FOREST(START_NODE);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(START_NODE);
  INSERT_FOREST(NUMBER_TO_LIST_NODE(ENTRY_LOCATION));
  INSERT_FOREST(TRANSITION_POINTER);
end TASK_ACCEPT;

procedure END_ACCEPT(CURRENT_LOCATION : in positive;
                     ENTRY_END        : in positive) is
-- post - The abstract grammar for the end of an accept statement is
--        generated.
CURRENT_NODE : LIST_NODE_POINTER;
LOOP_POINTER : SYMBOL_TABLE.SYM_TAB_ACCESS;
begin
  CURRENT_NODE := NUMBER_TO_LIST_NODE(CURRENT_LOCATION);
  if (not IS_COMPLETE) then
    INSERT_FOREST(CURRENT_NODE);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(CURRENT_NODE);
  INSERT_FOREST(TRANSITION_POINTER);
  INSERT_FOREST(NUMBER_TO_LIST_NODE(ENTRY_END));
end END_ACCEPT;

procedure EXPLICIT_END_ACCEPT(ENTRY_END : in positive) is
-- post - The abstract grammar for the end of an accept statement is
--        generated.
begin
  if (not IS_COMPLETE) then
    INSERT_FOREST(NUMBER_TO_LIST_NODE(ENTRY_END));
  end if;
end EXPLICIT_END_ACCEPT;

procedure GO_TO(CURRENT_LOCATION : in positive;
                GO_TO_LOCATION   : in SYMBOL_TABLE.SYM_TAB_ACCESS) is
-- post - The abstract grammar for a goto statement is generated.
START_NODE : LIST_NODE_POINTER;
begin
  START_NODE := NUMBER_TO_LIST_NODE(CURRENT_LOCATION);
  if (not IS_COMPLETE) then
```

```
    INSERT_FOREST(START_NODE);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(START_NODE);
  INSERT_FOREST(TRANSITION_POINTER);
  INSERT_FOREST(POINTER_TO_LIST_NODE(GO_TO_LOCATION));
  NEW_SYNTAX_LIST;
end GO_TO;

procedure END_LOOP(END_LOCATION : in positive;
                   LOOP_START   : in SYMBOL_TABLE.SYM_TAB_ACCESS) is
-- post - The abstract grammar for a loop is generated.
END_NODE : LIST_NODE_POINTER;
LOOP_POINTER : SYMBOL_TABLE.SYM_TAB_ACCESS;
begin
  END_NODE := NUMBER_TO_LIST_NODE(END_LOCATION);
  if (not IS_COMPLETE) then
    INSERT_FOREST(END_NODE);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(END_NODE);
  INSERT_FOREST(TRANSITION_POINTER);
  INSERT_FOREST(POINTER_TO_LIST_NODE(LOOP_START));
end END_LOOP;

procedure CONNECT_BLOCKS(CURRENT_LOCATION : in positive;
                         NEXT_LOCATION    : in positive) is
-- post - used to explicitly declare a transition between two known
--        code blocks.  The abstract grammar for a transition between
--        two petri net places is generated.
START_NODE : LIST_NODE_POINTER;
begin
  START_NODE := NUMBER_TO_LIST_NODE(CURRENT_LOCATION);
  if (not IS_COMPLETE) then
    INSERT_FOREST(START_NODE);
    NEW_SYNTAX_LIST;
  end if;
  INSERT_FOREST(START_NODE);
  INSERT_FOREST(TRANSITION_POINTER);
  INSERT_FOREST(NUMBER_TO_LIST_NODE(NEXT_LOCATION));
  NEW_SYNTAX_LIST;
end CONNECT_BLOCKS;

procedure EXPLICIT_END(NEXT_LOCATION : in positive) is
 -- post - The current forest is terminated and a new forest is begun.
begin
  if (not IS_COMPLETE) then
    INSERT_FOREST(NUMBER_TO_LIST_NODE(NEXT_LOCATION));
    NEW_SYNTAX_LIST;
  end if;
end EXPLICIT_END;
```

```
procedure TRANSLATE_TO_PEANUT is
-- post - used to translate the abstract petri net grammar to a
--        text file used as an input file to P-NUT petri net analyzer.
--        Produces two files: 1)  a.out -     P-NUT input file
--                            2)  place.dat - text file that describes all
--                                            the places that exist in the
--                                            petri net and/or the
--                                            places relation to the
--                                            original source code.
--        The net generator and code blocker are reset to their
--        initial states.
TRANSITION_NUMBER : positive := 1;
NET_FILE : TEXT_IO.file_type;
SYNTAX_LIST : ABSTRACT_SYNTAX_LIST.LIST;
INITIAL_MARK : LIST_NODE_POINTER;
PLACE_FILE : TEXT_IO.file_type;
START_SOURCE_INFO : TOKEN_SCANNER.SOURCE_RECORD;
STOP_SOURCE_INFO  : TOKEN_SCANNER.SOURCE_RECORD;
  function  POS_TO_LIT(NUMBER : string) return string is
  begin
    return (NUMBER(2..NUMBER'LAST));
  end POS_TO_LIT;
  procedure XLATE(SYNTAX_LIST : in out ABSTRACT_SYNTAX_LIST.LIST) is
    package PLACE_STACK is new GENERIC_STACK(LIST_NODE_POINTER);
  TEMP_POINTER : LIST_NODE_POINTER;
  PS : PLACE_STACK.STACK;
  SUCCESS : boolean;
  begin
    PLACE_STACK.CREATE(PS, SUCCESS);
    if (not SUCCESS) then
      raise NET_GENERATOR_OVERFLOW;
    end if;
    if (not ABSTRACT_SYNTAX_LIST.EMPTY(SYNTAX_LIST)) then
      ABSTRACT_SYNTAX_LIST.FIND_FIRST(SYNTAX_LIST);
      ABSTRACT_SYNTAX_LIST.RETRIEVE(SYNTAX_LIST, TEMP_POINTER);
      while (TEMP_POINTER.PETRI_TAG /= TRANSITION) loop
        PLACE_STACK.PUSH(PS, TEMP_POINTER);
        ABSTRACT_SYNTAX_LIST.FIND_NEXT(SYNTAX_LIST);
        ABSTRACT_SYNTAX_LIST.RETRIEVE(SYNTAX_LIST, TEMP_POINTER);
      end loop;
      ABSTRACT_SYNTAX_LIST.FIND_NEXT(SYNTAX_LIST); --skip transition pointer
      TEXT_IO.put(NET_FILE, ":t");
      TEXT_IO.put(NET_FILE, POS_TO_LIT(positive'IMAGE(TRANSITION_NUMBER)));
      TRANSITION_NUMBER := TRANSITION_NUMBER + 1;
      TEXT_IO.put(NET_FILE, ": ");
      PLACE_STACK.POP(PS, TEMP_POINTER);
      TEXT_IO.put(NET_FILE, "p");
      TEXT_IO.put(NET_FILE, POS_TO_LIT(positive'IMAGE(TEMP_POINTER.
                                                    SYMBOL.LOCATION)));
      while (not PLACE_STACK.EMPTY(PS)) loop
```

```
      PLACE_STACK.POP(PS, TEMP_POINTER);
      TEXT_IO.put(NET_FILE, ", p");
      TEXT_IO.put(NET_FILE, POS_TO_LIT(positive'IMAGE(TEMP_POINTER.
                                                    SYMBOL.LOCATION)));
    end loop;
    PLACE_STACK.DISPOSE(PS);
    TEXT_IO.put(NET_FILE, " -> ");
    ABSTRACT_SYNTAX_LIST.RETRIEVE(SYNTAX_LIST, TEMP_POINTER);
    TEXT_IO.put(NET_FILE, "p");
    TEXT_IO.put(NET_FILE, POS_TO_LIT(positive'IMAGE(TEMP_POINTER.
                                                  SYMBOL.LOCATION)));
    while (not ABSTRACT_SYNTAX_LIST.LAST(SYNTAX_LIST)) loop
      ABSTRACT_SYNTAX_LIST.FIND_NEXT(SYNTAX_LIST);
      ABSTRACT_SYNTAX_LIST.RETRIEVE(SYNTAX_LIST, TEMP_POINTER);
      TEXT_IO.put(NET_FILE, ", p");
      TEXT_IO.put(NET_FILE, POS_TO_LIT(positive'IMAGE(TEMP_POINTER.
                                                    SYMBOL.LOCATION)));
    end loop;
    TEXT_IO.new_line(NET_FILE);
  end if;
end XLATE;
begin
  begin
    TEXT_IO.create(NET_FILE, TEXT_IO.out_file, "a.out", "");
  exception
    when IO_EXCEPTIONS.USE_ERROR =>
      TEXT_IO.open(NET_FILE, TEXT_IO.out_file, "a.out", "");
    when others => raise;
  end;
  if (not FOREST_LIST.EMPTY(FOREST)) then
    XLATE(START_SYNTAX);
    FOREST_LIST.FIND_FIRST(FOREST);
    FOREST_LIST.RETRIEVE(FOREST, SYNTAX_LIST);
    XLATE(SYNTAX_LIST);
    while (not FOREST_LIST.LAST(FOREST)) loop
      FOREST_LIST.FIND_NEXT(FOREST);
      FOREST_LIST.RETRIEVE(FOREST, SYNTAX_LIST);
      XLATE(SYNTAX_LIST);
    end loop;
    ABSTRACT_SYNTAX_LIST.INSERT(STOP_PLACES, TRANSITION_POINTER);
    ABSTRACT_SYNTAX_LIST.INSERT(STOP_PLACES, CREATE_DUMMY_PLACE("STOP"));
    XLATE(STOP_PLACES);
    TEXT_IO.put(NET_FILE, "<p");
    ABSTRACT_SYNTAX_LIST.FIND_FIRST(START_SYNTAX);
    ABSTRACT_SYNTAX_LIST.RETRIEVE(START_SYNTAX, INITIAL_MARK);
    TEXT_IO.put(NET_FILE, POS_TO_LIT(positive'IMAGE(INITIAL_MARK.
                                                  SYMBOL.LOCATION)));
    TEXT_IO.put(NET_FILE, ">");
    TEXT_IO.close(NET_FILE);
  end if;
  begin
```

```
      TEXT_IO.create(PLACE_FILE, TEXT_IO.out_file, "place.dat", "");
    exception
      when IO_EXCEPTIONS.USE_ERROR =>
        TEXT_IO.open(PLACE_FILE, TEXT_IO.out_file, "place.dat", "");
      when others => raise;
    end;
    if (not CODE_BLOCKER.IS_CODE_BLOCK_LIST_CLEAR) then
      CODE_BLOCKER.FIND_FIRST_CODE_BLOCK;
      TEXT_IO.put(PLACE_FILE, "LOCATION");
      TEXT_IO.set_col(PLACE_FILE, 20);
      TEXT_IO.put(PLACE_FILE, "CODE_BLOCK_LABEL");
      TEXT_IO.set_col(PLACE_FILE, 50);
      TEXT_IO.put(PLACE_FILE, "STARTING LINE");
      TEXT_IO.set_col(PLACE_FILE, 65);
      TEXT_IO.put(PLACE_FILE, "ENDING LINE");
      TEXT_IO.new_line(PLACE_FILE, 2);
      loop
        TEXT_IO.put(PLACE_FILE, "p");
        TEXT_IO.put(PLACE_FILE, POS_TO_LIT(positive'IMAGE(CODE_BLOCKER.
                                            READ_CODE_BLOCK_NUMBER)));
        TEXT_IO.set_col(PLACE_FILE, 20);
        TEXT_IO.put(PLACE_FILE, CODE_BLOCKER.READ_CODE_BLOCK_LABEL);
        START_SOURCE_INFO := CODE_BLOCKER.READ_CODE_BLOCK_START;
        STOP_SOURCE_INFO  := CODE_BLOCKER.READ_CODE_BLOCK_STOP;
        TEXT_IO.set_col(PLACE_FILE, 55);
        TEXT_IO.put(PLACE_FILE, natural'IMAGE(START_SOURCE_INFO.LINE_NUMBER));
        TEXT_IO.set_col(PLACE_FILE, 70);
        TEXT_IO.put_line(PLACE_FILE, natural'IMAGE(STOP_SOURCE_INFO.
                                                          LINE_NUMBER));
        exit when CODE_BLOCKER.IS_LAST_CODE_BLOCK;
        CODE_BLOCKER.FIND_NEXT_CODE_BLOCK;
      end loop;
      TEXT_IO.close(PLACE_FILE);
      CODE_BLOCKER.CLEAR_CODE_BLOCKER;
      RESET_NET_GENERATOR;
    end if;
  end TRANSLATE_TO_PEANUT;

begin
  INITIALIZE_NET_GENERATOR;
end NET_GENERATOR;
```

APPENDIX E

"ADAFLOW" PROGRAM LISTING - SYMBOL TABLE

```
--***************************************************************--
--                                                               --
--  TITLE:          ADAFLOW                                      --
--                                                               --
--  MODULE NAME:    PACKAGE SYMBOL_TABLE                         --
--  FILE NAME:      SYM_TAB.ADS                                  --
--                                                               --
--  DATE CREATED:   01 MAR 88                                    --
--  LAST MODIFIED:  28 APR 88                                    --
--                                                               --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                     --
--                                                               --
--  DESCRIPTION:  This package contains the procedures which     --
--                define the interface to the symbol table.      --
--                                                               --
--***************************************************************--

with TOKEN_SCANNER;

package SYMBOL_TABLE is

  type SYMBOL_TAG is (OBJECT_DECLARATION_TAG,   TYPE_DECLARATION_TAG,
                      FUNCTION_DECLARATION_TAG, PROCEDURE_DECLARATION_TAG,
                      PACKAGE_DECLARATION_TAG,  TASK_DECLARATION_TAG,
                      ENTRY_TAG,
                      PACKAGE_BODY_TAG,         TASK_BODY_TAG,
                      ACCEPT_TAG,               LABEL_NAME,
                      SELECT_TAG,               LOOP_TAG);

  type SYM_TAB_RECORD is
    record
      NAME             : string(1..TOKEN_SCANNER.LINESIZE) := (others => ' ');
      NAME_LENGTH      : natural := 0;
      TAG_TYPE         : SYMBOL_TAG;
      LOCATION         : natural := 0; -- 0 indicates undeclared.
      REFERENCE_COUNT  : natural := 0; -- used to count the number of
    end record;                        -- pointers to this entry.  DO NOT
                                       -- COLLECT GARBAGE UNLESS THIS IS 1.

  type SYM_TAB_ACCESS is access SYM_TAB_RECORD;
```

```
SYMBOL_TABLE_OVERFLOW : exception;
DECLARATION_ERROR     : exception;
REFERENCE_ERROR       : exception;

procedure CLEAR_SYM_TAB;
-- post - SYM_TAB is returned to it's initialized state.

function  FULL_SYM_TAB return boolean;
-- post - If the size of SYM_TAB has not reached its bound then FULL is
--        FALSE else FULL is TRUE.

procedure EXIT_SCOPE;
-- post - SYM_TAB backs up one static nesting level.  The current entry is
--        defined as the entry that caused the corresponding scope entry to
--        occur.

procedure INSERT_SYM_TAB(KEY       : in string;
                         ATTRIBUTE : in SYMBOL_TAG;
                         LOCATION  : in natural);
-- pre  - SYM_TAB has not achieved its maximum allowable size.
-- post - If the ATTRUBUTE is OBJECT_DECLARATION_TAG, TYPE_CECLARATION_TAG,
--        or LABEL_NAME, a search is conducted at the local SNL for a
--        matching KEY. If no match is found, KEY is inserted with the given
--        attribute and location and is the the current entry, else no
--        action is taken and the current entry is the pre-existing entry
--        named by key.
--        If the ATTRIBUTE is FUNCTION_DECLARATION_TAG,
--        PROCEDURE_DECLARATION_TAG, PACKAGE_DECLARATION_TAG,
--        TASK_DECLARATION_TAG, or ENTRY_TAG, a search is conducted at the
--        local SNL for a matching KEY. If no match is found, KEY is inserted
--        with the given attribute and location and scope entry occurs, else
--        a check is made to see if the pre-existing entry is a
--        PROCEDURE_DECLARATION_TAG or a FUNCTION_DECLARATION_TAG.  If so,
--        location is updated and scope entry occurs.
--        If the ATTRIBUTE is PACKAGE_BODY_TAG, TASK_BODY_TAG, or
--        ACCEPT_TAG, the corresponding environment of definition is
--        located, the location updated, and then scope entry occurs.
--        If the ATTRIBUTE is LOOP_TAG or SELECT_TAG, the symbol is entered
--        with the given ATTRIBUTE and LOCATION and scope entry occurs.
-- exceptions raised - SYMBOL_TABLE_OVERFLOW if the symbol table's size
--                     has reached it's bound.
--                     DECLARATION_ERROR if the required environment of
--                     definition can not be found for a body declaration
--                     or if a declaration tag already exists at the current
--                     SNL.

function  FIND_KEY(KEY : in string) return SYM_TAB_ACCESS;
-- post - If the symbol table contains an entry whose key value is KEY,
--        then that entry is the current entry and FIND_KEY returns a
--        pointer to that symbol table record, else FIND_KEY returns
--        a null pointer and the current entry is undefined.  NOTE -
```

```
--        the symbol table IS case sensitive in it's comparison of keys and
--        the search is global in scope according to ADA visibility rules.

function  FIND_LOCAL_KEY(KEY : in string) return SYM_TAB_ACCESS;
-- post - If the symbol table contains an entry whose key value is KEY,
--        then that entry is the current entry and FIND_KEY returns a
--        pointer to that symbol table record, else FIND_KEY returns
--        a null pointer and the current entry is undefined.  NOTE -
--        the symbol table IS case sensitive in it's comparison of keys and
--        the search is local in scope according to ADA visibility rules.

function  FIND_SUBPROGRAM_END return SYM_TAB_ACCESS;
-- post - A search is conducted to find the parent enclosing subprogram
--        of the parse.  A pointer to the label "END" for this parent
--        enclosing subprogram is returned.  This function is used to
--        provide the operand for a "return" statement.  The current entry
--        is the corresponding end label for the enclosing subprogram of the
--        parse.
-- exceptions raised - REFERENCE_ERROR if no enclosing subprogram can be
--                     found or if a label "END" can not be found for
--                     an enclosing subprogram.

function  FIND_LOOP_END return SYM_TAB_ACCESS;
-- post - A search is conducted to find the enclosing loop
--        of the parse.  A pointer to the label "END" for this
--        enclosing loop is returned.  This function is used to
--        provide the operand for an "exit" statement.  The current entry
--        is the end label corresponding to the enclosing loop of the
--        parse.
-- exceptions raised - REFERENCE_ERROR if no enclosing loop can be
--                     found or if a label "END" can not be found for
--                     an enclosing loop.

function  FIND_TASK_END return SYM_TAB_ACCESS;
-- post - A search is conducted to find the enclosing task
--        of the parse.  A pointer to the label "END" for this
--        enclosing task is returned.  The current entry
--        is the end label corresponding to the enclosing task of the
--        parse.
-- exceptions raised - REFERENCE_ERROR if no enclosing task can be
--                     found or if a label "END" can not be found for
--                     an enclosing task.

procedure UPDATE_SYM_TAB(LOCATION : in natural);
-- pre  - The current entry is defined.
-- post - The current entry's location is changed to LOCATION.

function  SELECT_COMPONENT(KEY : in string) return SYM_TAB_ACCESS;
-- pre  - The current entry is defined.
-- post - SELECT_COMPONENT provides visibility to the next static nesting
--        level below the current entry.  If the symbol table contains an
```

```
--        entry whose key value is KEY at the next static nesting level,
--        then that entry is the current entry and FIND_KEY returns a
--        pointer to that symbol table record, else FIND_KEY returns
--        a null pointer and the current entry is undefined.  NOTE -
--        the symbol table IS case sensitive in it's comparison of keys.

function  RETRIEVE_SYM return SYM_TAB_ACCESS;
-- post - RETRIEVE_SYM returns a pointer to the current entry or null if
--        the current entry is undefined.

procedure SAVE_CURRENT_ENTRY;
-- pre  - The current entry is defined;
-- post - The current entry is saved in a last in first out data structure.

procedure RESTORE_CURRENT_ENTRY;
-- pre  - A current entry was saved;
-- post - The last current entry saved is the current entry.

procedure PRINT_SYMBOL_TABLE;
-- post - Useful as a debugging tool, PRINT_SYMBOL_TABLE prints a dump of
--        every symbol table entry, including attribute and location
--        information, to the standard output device.

end SYMBOL_TABLE;
```

```
--**************************************************************--
--                                                              --
--  TITLE:          ADAFLOW                                     --
--                                                              --
--  MODULE NAME:    PACKAGE SYMBOL_TABLE                        --
--  FILE NAME:      SYM_TAB.ADB                                 --
--                                                              --
--  DATE CREATED:   01 MAR 88                                   --
--  LAST MODIFIED:  28 APR 88                                   --
--                                                              --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                    --
--                                                              --
--  DESCRIPTION:  This package contains the procedures which    --
--                implement the interface to the symbol table.  --
--                                                              --
--**************************************************************--

with TOKEN_SCANNER,
     GENERIC_STACK,
     UNCHECKED_DEALLOCATION,
     TEXT_IO;

package body SYMBOL_TABLE is

  procedure FREE_SYM_REC is new
                      UNCHECKED_DEALLOCATION(SYM_TAB_RECORD,SYM_TAB_ACCESS);
  subtype DEFINITION_TAGS is SYMBOL_TAG range
                           FUNCTION_DECLARATION_TAG..ENTRY_TAG;
  subtype BODY_TAGS is SYMBOL_TAG range PACKAGE_BODY_TAG..ACCEPT_TAG;

  type LIST_NODE;

  type LIST_NODE_POINTER is access LIST_NODE;

  package SYMBOL_LIST is

    type LIST_INSTANCE is private;
    type LIST is access LIST_INSTANCE;

    LIST_OVERFLOW  : exception;
    LIST_UNDERFLOW : exception;

-- Operations:  If the list is not empty, then one of the nodes is designated
--   as the current node.  Ocaasionally, in the postcondition, it is necessary
--   to refer to the list of the current node as they were immediately before
--   execution of the operation.  L-pre and c-pre, respectively, are employed
--   for these references.

    procedure FIND_FIRST(L : in out LIST);
    -- pre  - The list L is not empty.
```

```
-- post - The first node is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure FIND_NEXT(L : in out LIST);
-- pre  - The list L is not empty and the last node is not the current node.
-- post - c-next in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
--                   - LIST_OVERFLOW  if the last node is the current node.

procedure FIND_PREVIOUS(L : in out LIST);
-- pre  - The list L is not empty and the first node is not the current node.
-- post - c-prior in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty or c is the first node.

procedure FIND_LAST(L : in out LIST);
-- pre  - The list L is not empty.
-- post - The last node in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure RETRIEVE(L : in LIST; ITEM : out LIST_NODE_POINTER);
-- pre  - The list L is not empty.
-- post - ITEM contains the value of the element in the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure UPDATE(L : in out LIST; ITEM : in LIST_NODE_POINTER);
-- pre  - The list L is not empty.
-- post - The current node in L contains ITEM as its element.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure INSERT(L : in out LIST; ITEM : in LIST_NODE_POINTER);
-- pre  - The number of nodes in L has not reached its bound.
-- post - A node containing ITEM is the last node in the list, and the last
--        node in L-pre, if any, is its predecessor.  The node containing
--        ITEM is the current node.
-- exceptions raised - LIST_OVERFLOW if L has reached its bound.

procedure DELETE(L : in out LIST);
-- pre  - The list L is not empty.
-- post - c-pre in not in the list L.  If c-pre was the first node,
--        then c-next, if it exists, is the successor of c-prior.  If the
--        list L is not empty, then the last node is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

function  SIZE_OF(L : in LIST) return natural;
-- post - SIZE_OF is the number of nodes in list L.

function  EMPTY(L : in LIST) return boolean;
-- post - If the list L has no nodes then EMPTY is true, else EMPTY is
--        false.
```

```
  function  FULL(L : in LIST) return boolean;
  -- post - If the number of nodes in the list L has reached the maximum
  --        allowed, then FULL is true, else FULL is false.

  function  FIRST(L : in LIST) return boolean;
  -- pre  - The list L is not empty.
  -- post - If the first node is the current node in L then FIRST is true, else
  --        FIRST is false.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.

  function  LAST(L : in LIST) return boolean;
  -- pre  - The list L is not empty.
  -- post - If the last node is the current node in L then LAST is true, else
  --        LAST is false.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.

  procedure CREATE(L : in out LIST; SUCCESS :  out boolean);
  -- post - If a list L can be created then L exists and is empty, and SUCCESS
  --        is TRUE else SUCCESS is FALSE.

  procedure DISPOSE(L : in out LIST);
  -- post - L-pre does not exist.

  procedure ASSIGN(L1 : in LIST; L2 : in out LIST);
  -- post - L2 contains the same nodes as L1.

  procedure SAVE_LIST(L : in LIST);
  -- post - L is saved in a last in first out data structure.

  procedure RESTORE_LIST(L : in out LIST);
  -- post - L is the last list that was saved.

private

  type NODE;
  type NODE_POINTER is access NODE;
  type NODE is
    record
      ELEMENT : LIST_NODE_POINTER;
      NEXT    : NODE_POINTER;
    end record;
  type LIST_INSTANCE is
    record
      HEAD    : NODE_POINTER := null;
      TAIL    : NODE_POINTER := null;
      CURRENT : NODE_POINTER := null;
      SIZE    : natural := 0;
    end record;

end SYMBOL_LIST;
```

```
type LIST_NODE is
  record
    SYMBOL          : SYM_TAB_ACCESS;
    SUB_LIST        : SYMBOL_LIST.LIST;
  end record;

SYM_TAB        : SYMBOL_LIST.LIST; -- the root of the symbol table tree
CURRENT_SNL    : SYMBOL_LIST.LIST; -- keeps track of the current branch
SEARCH_SNL     : SYMBOL_LIST.LIST; -- can be operated on without effecting
                                   -- the state of the symbol table.

LAST_FOUND : LIST_NODE_POINTER := null;

package STK_OF_LISTS is new GENERIC_STACK(SYMBOL_LIST.LIST);

SCOPE_STACK : STK_OF_LISTS.STACK;

package body SYMBOL_LIST is

  procedure FREE_NODE is new UNCHECKED_DEALLOCATION(NODE, NODE_POINTER);
  procedure FREE_LIST is new UNCHECKED_DEALLOCATION(LIST_INSTANCE, LIST);
  procedure FREE_SYM_REC is new
                   UNCHECKED_DEALLOCATION(SYM_TAB_RECORD,SYM_TAB_ACCESS);
  package STACK_LIST_INSTANCES is new GENERIC_STACK(LIST);

  SLI : STACK_LIST_INSTANCES.STACK;
  SUCCESS : boolean;

  procedure FIND_FIRST(L : in out LIST) is
  -- pre  - The list L is not empty.
  -- post - The first node is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    L.CURRENT := L.HEAD;
  end FIND_FIRST;

  procedure FIND_NEXT(L : in out LIST) is
  -- pre  - The list L is not empty and the last node is not the current node.
  -- post - c-next in L is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  --                   - LIST_OVERFLOW  if the last node is the current node.
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    if (LAST(L)) then
      raise LIST_OVERFLOW;
    end if;
```

```
  L.CURRENT := L.CURRENT.NEXT;
end FIND_NEXT;

procedure FIND_PREVIOUS(L : in out LIST) is
-- pre  - The list L is not empty and the first node is not the current node.
-- post - c-prior in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty or c is the first node.
TEMP_POINTER : NODE_POINTER;
begin
  if (EMPTY(L) or FIRST(L)) then
    raise LIST_UNDERFLOW;
  end if;
  TEMP_POINTER := L.HEAD;
  while (TEMP_POINTER.NEXT /= L.CURRENT) loop
    TEMP_POINTER := TEMP_POINTER.NEXT;
  end loop;
  L.CURRENT := TEMP_POINTER;
end FIND_PREVIOUS;

procedure FIND_LAST(L : in out LIST) is
-- pre  - The list L is not empty.
-- post - The last node in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  while (not LAST(L)) loop
    FIND_NEXT(L);
  end loop;
end FIND_LAST;

procedure RETRIEVE(L : in LIST; ITEM : out LIST_NODE_POINTER) is
-- pre  - The list L is not empty.
-- post - ITEM contains the value of the element in the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  ITEM := L.CURRENT.ELEMENT;
end RETRIEVE;

procedure UPDATE(L : in out LIST; ITEM : in LIST_NODE_POINTER) is
-- pre  - The list L is not empty.
-- post - The current node in L contains ITEM as its element.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
```

```
  L.CURRENT.ELEMENT := ITEM;
end UPDATE;

procedure INSERT(L : in out LIST; ITEM : in LIST_NODE_POINTER) is
-- pre  - The number of nodes in L has not reached its bound.
-- post - A node containing ITEM is the last node in the list, and the last
--        node in L-pre, if any, is its predecessor.  The node containing
--        ITEM is the current node.
-- exceptions raised - LIST_OVERFLOW if L has reached its bound.
TEMP_POINTER : NODE_POINTER;
begin
  if (FULL(L)) then
    raise LIST_OVERFLOW;
  end if;
  TEMP_POINTER := new NODE'(ITEM, null);
  TEMP_POINTER.ELEMENT.SYMBOL.REFERENCE_COUNT :=
    natural'SUCC(TEMP_POINTER.ELEMENT.SYMBOL.REFERENCE_COUNT);
  if (L.HEAD = null) then
    L.HEAD := TEMP_POINTER;
    L.TAIL := TEMP_POINTER;
  else
    L.TAIL.NEXT := TEMP_POINTER;
    L.TAIL      := TEMP_POINTER;
  end if;
  L.CURRENT := TEMP_POINTER;
  L.SIZE := L.SIZE + 1;
end INSERT;

procedure DELETE(L : in out LIST) is
-- pre  - The list L is not empty.
-- post - c-pre in not in the list L.  If c-pre was the first node,
--        then c-next, if it exists, is the successor of c-prior.  If the
--        list L is not empty, then the last node is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
TEMP_POINTER : NODE_POINTER;
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  if (L.CURRENT /= L.HEAD) then
    TEMP_POINTER := L.HEAD;
    while (TEMP_POINTER.NEXT /= L.CURRENT) loop
      TEMP_POINTER := TEMP_POINTER.NEXT;
    end loop;
    TEMP_POINTER.NEXT := L.CURRENT.NEXT;
    if (L.CURRENT = L.TAIL) then
      L.TAIL := TEMP_POINTER;
    end if;
  else
    if (L.HEAD = L.TAIL) then
      L.TAIL := null;
```

```
    end if;
    L.HEAD := L.HEAD.NEXT;
  end if;
  if (L.CURRENT.ELEMENT.SYMBOL.REFERENCE_COUNT > 1) then
    L.CURRENT.ELEMENT.SYMBOL.REFERENCE_COUNT :=
      positive'PRED(L.CURRENT.ELEMENT.SYMBOL.REFERENCE_COUNT);
  else
    FREE_SYM_REC(L.CURRENT.ELEMENT.SYMBOL);
  end if;
  DISPOSE(L.CURRENT.ELEMENT.SUB_LIST);
  FREE_NODE(L.CURRENT);
  L.CURRENT := L.TAIL;
  L.SIZE := L.SIZE - 1;
end DELETE;

function  SIZE_OF(L : in LIST) return natural is
-- post - SIZE_OF is the number of nodes in list L.
begin
  return (L.SIZE);
end SIZE_OF;

function  EMPTY(L : in LIST) return boolean is
-- post - If the list L has no nodes then EMPTY is true, else EMPTY is
--        false.
begin
  return (L.HEAD = null);
end EMPTY;

function  FULL(L : in LIST) return boolean is
-- post - If the number of nodes in the list L has reached the maximum
--        allowed, then FULL is true, else FULL is false.
TEMP_POINTER : NODE_POINTER;
begin
  TEMP_POINTER := new NODE;
  FREE_NODE(TEMP_POINTER);
  return (FALSE);
exception
  when STORAGE_ERROR =>
    return (TRUE);
  when others =>
    raise;
end FULL;

function  FIRST(L : in LIST) return boolean is
-- pre  - The list L is not empty.
-- post - If the first node is the current node in L then FIRST is true, else
--        FIRST is false.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
```

```
    end if;
    return (L.CURRENT = L.HEAD);
  end FIRST;

  function  LAST(L : in LIST) return boolean is
  -- pre  - The list L is not empty.
  -- post - If the last node is the current node in L then LAST is true, else
  --        LAST is false.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    return (L.CURRENT = L.TAIL);
  end LAST;

  procedure CREATE(L : in out LIST; SUCCESS :  out boolean) is
  -- post - If a list L can be created then L exists and is empty, and SUCCESS
  --        is TRUE else SUCCESS is FALSE.
  begin
    L := new LIST_INSTANCE'(null, null, null, 0);
    SUCCESS := TRUE;
  exception
    when STORAGE_ERROR =>
      SUCCESS := FALSE;
    when others =>
      raise;
  end CREATE;

  procedure DISPOSE(L : in out LIST) is
  -- post - L-pre does not exist.
  begin
    if (not EMPTY(L)) then
      FIND_LAST(L);
      while (not EMPTY(L)) loop
        DELETE(L);
      end loop;
    end if;
    FREE_LIST(L);
  end DISPOSE;

  procedure ASSIGN(L1 : in LIST; L2 : in out LIST) is
  -- post - L2 contains the same nodes as L1.
  begin
    L2.HEAD    := L1.HEAD;
    L2.CURRENT := L1.CURRENT;
    L2.TAIL    := L1.TAIL;
    L2.SIZE    := L1.SIZE;
  end ASSIGN;
```

```
  procedure SAVE_LIST(L : in LIST) is
  -- post - L is saved in a last in first out data structure.
  TEMP_LIST : LIST;
  SUCCESS   : boolean;
  begin
    CREATE(TEMP_LIST, SUCCESS);
    if (not SUCCESS) then
      raise SYMBOL_TABLE_OVERFLOW;
    end if;
    ASSIGN(L, TEMP_LIST);
    STACK_LIST_INSTANCES.PUSH(SLI, TEMP_LIST);
  end SAVE_LIST;

  procedure RESTORE_LIST(L : in out LIST) is
  -- post - L is the last list that was saved.
  TEMP_LIST : LIST;
  begin
    STACK_LIST_INSTANCES.POP(SLI, TEMP_LIST);
    ASSIGN(TEMP_LIST, L);
    FREE_LIST(TEMP_LIST);
  end RESTORE_LIST;

begin
  STACK_LIST_INSTANCES.CREATE(SLI, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
end SYMBOL_LIST;

function SNL_SEARCH(KEY : in string) return LIST_NODE_POINTER is
-- post - If the symbol table contains an entry at the local scope whose
--        key value is KEY, then that entry is the current entry in the
--        list SEARCH_SNL and SNL_SEARCH returns a pointer to that list
--        node, else SNL_SEARCH returns a null pointer and the
--        current entry in the list SEARCH_SNL is the last entry.
SEARCH_POINTER : LIST_NODE_POINTER;
begin
  if (SYMBOL_LIST.EMPTY(SEARCH_SNL)) then
    return (null);
  else
    SYMBOL_LIST.FIND_FIRST(SEARCH_SNL);
    loop
      SYMBOL_LIST.RETRIEVE(SEARCH_SNL, SEARCH_POINTER);
      if ((SEARCH_POINTER.SYMBOL.NAME_LENGTH = KEY'LENGTH) and then
          (SEARCH_POINTER.SYMBOL.NAME(1..KEY'LAST) = KEY)) then
        return (SEARCH_POINTER);
      else
        exit when (SYMBOL_LIST.LAST(SEARCH_SNL));
        SYMBOL_LIST.FIND_NEXT(SEARCH_SNL);
      end if;
    end loop;
```

```
    return (null);
  end if;
end SNL_SEARCH;

procedure INITIALIZE_SYM_TAB is
-- post - SYM_TAB contains the names and defined attributes for the language
--        defined enclosing scopes.
SUCCESS : boolean;
begin
  SYMBOL_LIST.CREATE(SYM_TAB, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
  SYMBOL_LIST.CREATE(SEARCH_SNL, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
  STK_OF_LISTS.CREATE(SCOPE_STACK, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
  CURRENT_SNL := SYM_TAB;
end INITIALIZE_SYM_TAB;

procedure CLEAR_SYM_TAB is
-- post - SYM_TAB is returned to it's initialized state.
SUCCESS : boolean;
begin
  SYMBOL_LIST.DISPOSE(SYM_TAB);
  STK_OF_LISTS.DISPOSE(SCOPE_STACK);
  SYMBOL_LIST.CREATE(SYM_TAB, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
  STK_OF_LISTS.CREATE(SCOPE_STACK, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
  CURRENT_SNL := SYM_TAB;
  LAST_FOUND := null;
end CLEAR_SYM_TAB;

function  FULL_SYM_TAB return boolean is
-- post - If the size of SYM_TAB has not reached its bound then FULL is
--        FALSE else FULL is TRUE.
begin
  return (SYMBOL_LIST.FULL(CURRENT_SNL));
end FULL_SYM_TAB;

procedure ENTER_SCOPE is
-- post - SYM_TAB enters the next static nesting level.
```

```
TEMP_POINTER : LIST_NODE_POINTER;
begin
  STK_OF_LISTS.PUSH(SCOPE_STACK, CURRENT_SNL);
  SYMBOL_LIST.RETRIEVE(SEARCH_SNL, TEMP_POINTER);
  CURRENT_SNL := TEMP_POINTER.SUB_LIST;
  SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
end ENTER_SCOPE;

procedure ENTER_SEARCH_SCOPE is
-- post - SYM_TAB enters the next static nesting level.
TEMP_POINTER : LIST_NODE_POINTER;
begin
  SYMBOL_LIST.RETRIEVE(SEARCH_SNL, TEMP_POINTER);
  SYMBOL_LIST.ASSIGN(TEMP_POINTER.SUB_LIST, SEARCH_SNL);
end ENTER_SEARCH_SCOPE;

procedure EXIT_SCOPE is
-- post - SYM_TAB backs up one static nesting level.  The current entry is
--        defined as the entry that caused the corresponding scope entry to
--        occur.
TEMP_POINTER : LIST_NODE_POINTER;
begin
  STK_OF_LISTS.POP(SCOPE_STACK, CURRENT_SNL);
  SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
  SYMBOL_LIST.RETRIEVE(SEARCH_SNL, LAST_FOUND);
end EXIT_SCOPE;

procedure INSERT_SYM_TAB(KEY       : in string;
                         ATTRIBUTE : in SYMBOL_TAG;
                         LOCATION  : in natural) is
-- pre  - SYM_TAB has not achieved its maximum allowable size.
-- post - If the ATTRUBUTE is OBJECT_DECLARATION_TAG, TYPE_CECLARATION_TAG,
--        or LABEL_NAME, a search is conducted at the local SNL for a
--        matching KEY. If no match is found, KEY is inserted with the given
--        attribute and location and is the the current entry, else no
--        action is taken and the current entry is the pre-existing entry
--        named by key.
--        If the ATTRIBUTE is FUNCTION_DECLARATION_TAG,
--        PROCEDURE_DECLARATION_TAG, PACKAGE_DECLARATION_TAG,
--        TASK_DECLARATION_TAG, or ENTRY_TAG, a search is conducted at the
--        local SNL for a matching KEY. If no match is found, KEY is inserted
--        with the given attribute and location and scope entry occurs, else
--        a check is made to see if the pre-existing entry is a
--        PROCEDURE_DECLARATION_TAG or a FUNCTION_DECLARATION_TAG.  If so,
--        location is updated and scope entry occurs.
--        If the ATTRIBUTE is PACKAGE_BODY_TAG, TASK_BODY_TAG, or
--        ACCEPT_TAG, the corresponding environment of definition is
--        located, the location updated, and then scope entry occurs.
--        If the ATTRIBUTE is LOOP_TAG or SELECT_TAG, the symbol is entered
--        with the given ATTRIBUTE and LOCATION and scope entry occurs.
-- exceptions raised - SYMBOL_TABLE_OVERFLOW if the symbol table's size
```

```
--                      has reached it's bound.
--                      DECLARATION_ERROR if the required environment of
--                      definition can not be found for a body declaration
--                      or if a declaration tag already exists at the current
--                      SNL.
TEMP_POINTER   : LIST_NODE_POINTER;
SEARCH_POINTER : LIST_NODE_POINTER;
TEMP_SYMBOL    : SYM_TAB_ACCESS;
SUCCESS : boolean;
use SYMBOL_LIST;
begin
  if ((ATTRIBUTE = OBJECT_DECLARATION_TAG) or else
  (ATTRIBUTE = TYPE_DECLARATION_TAG) or else (ATTRIBUTE = LABEL_NAME)) then
    SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
    SEARCH_POINTER := SNL_SEARCH(KEY);
    if (SEARCH_POINTER = null) then
      if (not SYMBOL_LIST.FULL(CURRENT_SNL)) then
        TEMP_POINTER := new LIST_NODE;
        TEMP_POINTER.SYMBOL := new SYM_TAB_RECORD;
        TEMP_POINTER.SYMBOL.NAME_LENGTH := KEY'LENGTH;
        TEMP_POINTER.SYMBOL.NAME := (others => ' ');
        TEMP_POINTER.SYMBOL.NAME(1..KEY'LAST) := KEY;
        TEMP_POINTER.SYMBOL.TAG_TYPE := ATTRIBUTE;
        TEMP_POINTER.SYMBOL.LOCATION := LOCATION;
        TEMP_POINTER.SYMBOL.REFERENCE_COUNT := 0;
        SYMBOL_LIST.CREATE(TEMP_POINTER.SUB_LIST, SUCCESS);
        if (not SUCCESS) then
          raise SYMBOL_TABLE_OVERFLOW;
        end if;
        SYMBOL_LIST.INSERT(CURRENT_SNL, TEMP_POINTER);
        SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
        LAST_FOUND := TEMP_POINTER;
      else
        raise SYMBOL_TABLE_OVERFLOW;
      end if;
    else
      SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
      LAST_FOUND := SEARCH_POINTER;
    end if;
  elsif (ATTRIBUTE in DEFINITION_TAGS) then
    SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
    SEARCH_POINTER := SNL_SEARCH(KEY);
    if (SEARCH_POINTER = null) then
      if (not SYMBOL_LIST.FULL(CURRENT_SNL)) then
        TEMP_POINTER := new LIST_NODE;
        TEMP_POINTER.SYMBOL := new SYM_TAB_RECORD;
        TEMP_POINTER.SYMBOL.NAME_LENGTH := KEY'LENGTH;
        TEMP_POINTER.SYMBOL.NAME := (others => ' ');
        TEMP_POINTER.SYMBOL.NAME(1..KEY'LAST) := KEY;
        TEMP_POINTER.SYMBOL.TAG_TYPE := ATTRIBUTE;
        TEMP_POINTER.SYMBOL.LOCATION := LOCATION;
```

```
      TEMP_POINTER.SYMBOL.REFERENCE_COUNT := 0;
      SYMBOL_LIST.CREATE(TEMP_POINTER.SUB_LIST, SUCCESS);
      if (not SUCCESS) then
        raise SYMBOL_TABLE_OVERFLOW;
      end if;
      SYMBOL_LIST.INSERT(CURRENT_SNL, TEMP_POINTER);
      SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
      LAST_FOUND := TEMP_POINTER;
      ENTER_SCOPE;
    else
      raise SYMBOL_TABLE_OVERFLOW;
    end if;
  elsif ((ATTRIBUTE = FUNCTION_DECLARATION_TAG) or
         (ATTRIBUTE = PROCEDURE_DECLARATION_TAG)) then
    UPDATE_SYM_TAB(LOCATION);
    SYMBOL_LIST.ASSIGN(SEARCH_SNL, CURRENT_SNL);
    LAST_FOUND := SEARCH_POINTER;
    ENTER_SCOPE;
  else
    raise DECLARATION_ERROR;
  end if;
elsif (ATTRIBUTE in BODY_TAGS) then
  SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
  TEMP_SYMBOL := FIND_KEY(KEY);
  if (TEMP_SYMBOL = null) then
    LAST_FOUND := null;
    raise DECLARATION_ERROR;
  else
    UPDATE_SYM_TAB(LOCATION);
    if (SEARCH_SNL = CURRENT_SNL) then
      SYMBOL_LIST.ASSIGN(SEARCH_SNL, CURRENT_SNL);
    end if;
    SYMBOL_LIST.RETRIEVE(SEARCH_SNL, LAST_FOUND);
    ENTER_SCOPE;
  end if;
elsif ((ATTRIBUTE = LOOP_TAG) or else (ATTRIBUTE = SELECT_TAG)) then
  if (not SYMBOL_LIST.FULL(CURRENT_SNL)) then
    TEMP_POINTER := new LIST_NODE;
    TEMP_POINTER.SYMBOL := new SYM_TAB_RECORD;
    TEMP_POINTER.SYMBOL.NAME_LENGTH := KEY'LENGTH;
    TEMP_POINTER.SYMBOL.NAME := (others => ' ');
    TEMP_POINTER.SYMBOL.NAME(1..KEY'LAST) := KEY;
    TEMP_POINTER.SYMBOL.TAG_TYPE := ATTRIBUTE;
    TEMP_POINTER.SYMBOL.LOCATION := LOCATION;
    TEMP_POINTER.SYMBOL.REFERENCE_COUNT := 0;
    SYMBOL_LIST.CREATE(TEMP_POINTER.SUB_LIST, SUCCESS);
    if (not SUCCESS) then
      raise SYMBOL_TABLE_OVERFLOW;
    end if;
    SYMBOL_LIST.INSERT(CURRENT_SNL, TEMP_POINTER);
    SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
```

```
      LAST_FOUND := TEMP_POINTER;
      ENTER_SCOPE;
    else
      raise SYMBOL_TABLE_OVERFLOW;
    end if;
  end if;
exception
  when STORAGE_ERROR =>
    raise SYMBOL_TABLE_OVERFLOW;
  when others =>
    raise;
end INSERT_SYM_TAB;

function FIND_KEY(KEY : in string) return SYM_TAB_ACCESS is
-- post - If the symbol table contains an entry whose key value is KEY,
--        then that entry is the current entry and FIND_KEY returns a
--        pointer to that symbol table record, else FIND_KEY returns
--        a null pointer and the current entry is undefined.  NOTE -
--        the symbol table IS case sensitive in it's comparison of keys and
--        the search is global in scope according to ADA visibility rules.
TEMP_POINTER : LIST_NODE_POINTER;
TEMP_LIST    : SYMBOL_LIST.LIST;
SEARCH_STACK : STK_OF_LISTS.STACK;
SUCCESS : boolean;
begin
  STK_OF_LISTS.CREATE(SEARCH_STACK, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
  SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
  TEMP_POINTER := SNL_SEARCH(KEY);
  if (TEMP_POINTER /= null) then
    LAST_FOUND := TEMP_POINTER;
    return (TEMP_POINTER.SYMBOL);
  else
    while (not STK_OF_LISTS.EMPTY(SCOPE_STACK)) loop
      STK_OF_LISTS.POP(SCOPE_STACK, TEMP_LIST);
      STK_OF_LISTS.PUSH(SEARCH_STACK, TEMP_LIST);
      SYMBOL_LIST.ASSIGN(TEMP_LIST, SEARCH_SNL);
      TEMP_POINTER := SNL_SEARCH(KEY);
      if (TEMP_POINTER /= null) then
        while (not STK_OF_LISTS.EMPTY(SEARCH_STACK)) loop
          STK_OF_LISTS.POP(SEARCH_STACK, TEMP_LIST);
          STK_OF_LISTS.PUSH(SCOPE_STACK, TEMP_LIST);
        end loop;
        LAST_FOUND := TEMP_POINTER;
        return (TEMP_POINTER.SYMBOL);
      end if;
    end loop;
    while (not STK_OF_LISTS.EMPTY(SEARCH_STACK)) loop
      STK_OF_LISTS.POP(SEARCH_STACK, TEMP_LIST);
```

```
      STK_OF_LISTS.PUSH(SCOPE_STACK, TEMP_LIST);
    end loop;
    LAST_FOUND := null;
    return (null);
  end if;
end FIND_KEY;

function FIND_LOCAL_KEY(KEY : in string) return SYM_TAB_ACCESS is
-- post - If the symbol table contains an entry whose key value is KEY,
--        then that entry is the current entry and FIND_KEY returns a
--        pointer to that symbol table record, else FIND_KEY returns
--        a null pointer and the current entry is undefined.  NOTE -
--        the symbol table IS case sensitive in it's comparison of keys and
--        the search is local in scope according to ADA visibility rules.
TEMP_POINTER : LIST_NODE_POINTER;
begin
  SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
  TEMP_POINTER := SNL_SEARCH(KEY);
  if (TEMP_POINTER /= null) then
    SYMBOL_LIST.ASSIGN(SEARCH_SNL, CURRENT_SNL);
    LAST_FOUND := TEMP_POINTER;
    return (TEMP_POINTER.SYMBOL);
  else
    LAST_FOUND := null;
    return (null);
  end if;
end FIND_LOCAL_KEY;

function  FIND_SUBPROGRAM_END return SYM_TAB_ACCESS is
-- post - A search is conducted to find the parent enclosing subprogram
--        of the parse.  A pointer to the label "END" for this parent
--        enclosing subprogram is returned.  This function is used to
--        provide the operand for a "return" statement.  The current entry
--        is the end label corresponding to the enclosing subprogram of the
--        parse.
-- exceptions raised - REFERENCE_ERROR if no enclosing subprogram can be
--                     found or if a label "END" can not be found for
--                     an enclosing subprogram.
PARENT     : LIST_NODE_POINTER;
TEMP_LIST  : SYMBOL_LIST.LIST;
SEARCH_STACK : STK_OF_LISTS.STACK;
SUCCESS : boolean;
begin
  STK_OF_LISTS.CREATE(SEARCH_STACK, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
  SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
  if (not STK_OF_LISTS.EMPTY(SCOPE_STACK)) then
    STK_OF_LISTS.POP(SCOPE_STACK, TEMP_LIST);
    STK_OF_LISTS.PUSH(SEARCH_STACK, TEMP_LIST);
```

```
    SYMBOL_LIST.ASSIGN(TEMP_LIST, SEARCH_SNL);
    SYMBOL_LIST.RETRIEVE(SEARCH_SNL, PARENT);
    while ((PARENT.SYMBOL.TAG_TYPE /= FUNCTION_DECLARATION_TAG) and then
    (PARENT.SYMBOL.TAG_TYPE /= PROCEDURE_DECLARATION_TAG)) loop
      if (STK_OF_LISTS.EMPTY(SCOPE_STACK)) then
        raise REFERENCE_ERROR;
      end if;
      STK_OF_LISTS.POP(SCOPE_STACK, TEMP_LIST);
      STK_OF_LISTS.PUSH(SEARCH_STACK, TEMP_LIST);
      SYMBOL_LIST.ASSIGN(TEMP_LIST, SEARCH_SNL);
      SYMBOL_LIST.RETRIEVE(SEARCH_SNL, PARENT);
    end loop;
    while (not STK_OF_LISTS.EMPTY(SEARCH_STACK)) loop
      STK_OF_LISTS.POP(SEARCH_STACK, TEMP_LIST);
      STK_OF_LISTS.PUSH(SCOPE_STACK, TEMP_LIST);
    end loop;
    SYMBOL_LIST.ASSIGN(PARENT.SUB_LIST, SEARCH_SNL);
    PARENT := SNL_SEARCH("END");
    if (PARENT /= null) then
      LAST_FOUND := PARENT;
      return (PARENT.SYMBOL);
    else
      raise REFERENCE_ERROR;
    end if;
  else
    raise REFERENCE_ERROR;
  end if;
end FIND_SUBPROGRAM_END;

function  FIND_LOOP_END return SYM_TAB_ACCESS is
-- post - A search is conducted to find the enclosing loop
--        of the parse.  A pointer to the label "END" for this
--        enclosing loop is returned.  This function is used to
--        provide the operand for an "exit" statement.  The current entry
--        is the end label corresponding to the enclosing loop of the
--        parse.
-- exceptions raised - REFERENCE_ERROR if no enclosing loop can be
--                     found or if a label "END" can not be found for
--                     an enclosing loop.
PARENT    : LIST_NODE_POINTER;
TEMP_LIST : SYMBOL_LIST.LIST;
SEARCH_STACK : STK_OF_LISTS.STACK;
SUCCESS : boolean;
begin
  STK_OF_LISTS.CREATE(SEARCH_STACK, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
  SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
  if (not STK_OF_LISTS.EMPTY(SCOPE_STACK)) then
    STK_OF_LISTS.POP(SCOPE_STACK, TEMP_LIST);
```

```
    STK_OF_LISTS.PUSH(SEARCH_STACK, TEMP_LIST);
    SYMBOL_LIST.ASSIGN(TEMP_LIST, SEARCH_SNL);
    SYMBOL_LIST.RETRIEVE(SEARCH_SNL, PARENT);
    while (PARENT.SYMBOL.TAG_TYPE /= LOOP_TAG) loop
      if (STK_OF_LISTS.EMPTY(SCOPE_STACK)) then
        raise REFERENCE_ERROR;
      end if;
      STK_OF_LISTS.POP(SCOPE_STACK, TEMP_LIST);
      STK_OF_LISTS.PUSH(SEARCH_STACK, TEMP_LIST);
      SYMBOL_LIST.ASSIGN(TEMP_LIST, SEARCH_SNL);
      SYMBOL_LIST.RETRIEVE(SEARCH_SNL, PARENT);
    end loop;
    while (not STK_OF_LISTS.EMPTY(SEARCH_STACK)) loop
      STK_OF_LISTS.POP(SEARCH_STACK, TEMP_LIST);
      STK_OF_LISTS.PUSH(SCOPE_STACK, TEMP_LIST);
    end loop;
    SYMBOL_LIST.ASSIGN(PARENT.SUB_LIST, SEARCH_SNL);
    PARENT := SNL_SEARCH("END");
    if (PARENT /= null) then
      LAST_FOUND := PARENT;
      return (PARENT.SYMBOL);
    else
      raise REFERENCE_ERROR;
    end if;
  else
    raise REFERENCE_ERROR;
  end if;
end FIND_LOOP_END;

function  FIND_TASK_END return SYM_TAB_ACCESS is
-- post - A search is conducted to find the enclosing task
--        of the parse.  A pointer to the label "END" for this
--        enclosing task is returned.  The current entry
--        is the end label corresponding to the enclosing task of the
--        parse.
-- exceptions raised - REFERENCE_ERROR if no enclosing task can be
--                     found or if a label "END" can not be found for
--                     an enclosing task.
PARENT    : LIST_NODE_POINTER;
TEMP_LIST : SYMBOL_LIST.LIST;
SEARCH_STACK : STK_OF_LISTS.STACK;
SUCCESS : boolean;
begin
  STK_OF_LISTS.CREATE(SEARCH_STACK, SUCCESS);
  if (not SUCCESS) then
    raise SYMBOL_TABLE_OVERFLOW;
  end if;
  SYMBOL_LIST.ASSIGN(CURRENT_SNL, SEARCH_SNL);
  if (not STK_OF_LISTS.EMPTY(SCOPE_STACK)) then
    STK_OF_LISTS.POP(SCOPE_STACK, TEMP_LIST);
    STK_OF_LISTS.PUSH(SEARCH_STACK, TEMP_LIST);
```

```
    SYMBOL_LIST.ASSIGN(TEMP_LIST, SEARCH_SNL);
    SYMBOL_LIST.RETRIEVE(SEARCH_SNL, PARENT);
    while (PARENT.SYMBOL.TAG_TYPE /= TASK_DECLARATION_TAG) loop
      if (STK_OF_LISTS.EMPTY(SCOPE_STACK)) then
        raise REFERENCE_ERROR;
      end if;
      STK_OF_LISTS.POP(SCOPE_STACK, TEMP_LIST);
      STK_OF_LISTS.PUSH(SEARCH_STACK, TEMP_LIST);
      SYMBOL_LIST.ASSIGN(TEMP_LIST, SEARCH_SNL);
      SYMBOL_LIST.RETRIEVE(SEARCH_SNL, PARENT);
    end loop;
    while (not STK_OF_LISTS.EMPTY(SEARCH_STACK)) loop
      STK_OF_LISTS.POP(SEARCH_STACK, TEMP_LIST);
      STK_OF_LISTS.PUSH(SCOPE_STACK, TEMP_LIST);
    end loop;
    SYMBOL_LIST.ASSIGN(PARENT.SUB_LIST, SEARCH_SNL);
    PARENT := SNL_SEARCH("END");
    if (PARENT /= null) then
      LAST_FOUND := PARENT;
      return (PARENT.SYMBOL);
    else
      raise REFERENCE_ERROR;
    end if;
  else
    raise REFERENCE_ERROR;
  end if;
end FIND_TASK_END;

procedure UPDATE_SYM_TAB(LOCATION : in natural) is
-- pre  - The current entry is defined.
-- post - The current entry's location is changed to LOCATION.
TEMP_POINTER : LIST_NODE_POINTER;
begin
  SYMBOL_LIST.RETRIEVE(SEARCH_SNL, TEMP_POINTER);
  TEMP_POINTER.SYMBOL.LOCATION := LOCATION;
  SYMBOL_LIST.UPDATE(SEARCH_SNL, TEMP_POINTER);
end UPDATE_SYM_TAB;

function SELECT_COMPONENT(KEY : in string) return SYM_TAB_ACCESS is
-- pre  - FIND_KEY or SELECT_COMPONENT returns a non-null value.
-- post - SELECT_COMPONENT provides visibility to the next static nesting
--        level below the current entry.
--        If the symbol table contains an entry whose key value is KEY,
--        then that entry is the current entry and FIND_KEY returns a
--        pointer to that symbol table record, else FIND_KEY returns
--        a null pointer and the current entry is undefined.  NOTE -
--        The symbol table IS case sensitive in it's comparison of keys.
TEMP_POINTER : LIST_NODE_POINTER;
begin
  ENTER_SEARCH_SCOPE;
  TEMP_POINTER := SNL_SEARCH(KEY);
```

```
  if (TEMP_POINTER = null) then
    LAST_FOUND := null;
    return (null);
  else
    LAST_FOUND := TEMP_POINTER;
    return (TEMP_POINTER.SYMBOL);
  end if;
end SELECT_COMPONENT;

function  RETRIEVE_SYM return SYM_TAB_ACCESS is
-- post - RETRIEVE_SYM returns a pointer to the current entry or null if
--         the current entry is undefined.
TEMP_POINTER : LIST_NODE_POINTER;
begin
  if (LAST_FOUND /= null) then
    return (LAST_FOUND.SYMBOL);
  else
    return (null);
  end if;
end RETRIEVE_SYM;

procedure SAVE_CURRENT_ENTRY is
-- pre  - The current entry is defined;
-- post - The current entry is saved in a last in first out data structure.
begin
  SYMBOL_LIST.SAVE_LIST(SEARCH_SNL);
end SAVE_CURRENT_ENTRY;

procedure RESTORE_CURRENT_ENTRY is
-- pre  - A current entry was saved;
-- post - The last current entry saved is the current entry.
begin
  SYMBOL_LIST.RESTORE_LIST(SEARCH_SNL);
  SYMBOL_LIST.RETRIEVE(SEARCH_SNL, LAST_FOUND);
end RESTORE_CURRENT_ENTRY;

procedure PRINT_SYMBOL_TABLE is
-- post - Useful as a debugging tool, PRINT_SYMBOL_TABLE prints a dump of
--         every symbol table entry, including attribute and location
--         information, to the standard output device.  The current entry is
--         undefined.
TEMP_POINTER : LIST_NODE_POINTER;
SEARCH_STACK : STK_OF_LISTS.STACK;
TEMP_LIST    : SYMBOL_LIST.LIST;
SUCCESS : boolean;
  procedure PRINT_RECORD(SP : in SYM_TAB_ACCESS) is
  use TEXT_IO;
  begin
    new_line;
    for INDEX in 1..SP.NAME_LENGTH loop
      put(SP.NAME(INDEX));
```

```
      end loop;
      set_col(30);
      put(SYMBOL_TAG'IMAGE(SP.TAG_TYPE));
      set_col(60);
      put_line(natural'IMAGE(SP.LOCATION));
    end PRINT_RECORD;
  begin
    STK_OF_LISTS.CREATE(SEARCH_STACK, SUCCESS);
    if (not SUCCESS) then
      raise SYMBOL_TABLE_OVERFLOW;
    end if;
    if (not SYMBOL_LIST.EMPTY(SYM_TAB)) then
      SYMBOL_LIST.FIND_FIRST(SYM_TAB);
      TEMP_LIST := SYM_TAB;
      loop
        while (not SYMBOL_LIST.EMPTY(TEMP_LIST)) loop
          STK_OF_LISTS.PUSH(SEARCH_STACK, TEMP_LIST);
          SYMBOL_LIST.RETRIEVE(TEMP_LIST, TEMP_POINTER);
          TEMP_LIST := TEMP_POINTER.SUB_LIST;
          if (not SYMBOL_LIST.EMPTY(TEMP_LIST)) then
            SYMBOL_LIST.FIND_FIRST(TEMP_LIST);
          end if;
        end loop;
        STK_OF_LISTS.POP(SEARCH_STACK, TEMP_LIST);
        SYMBOL_LIST.RETRIEVE(TEMP_LIST, TEMP_POINTER);
        PRINT_RECORD(TEMP_POINTER.SYMBOL);
        if (not SYMBOL_LIST.LAST(TEMP_LIST)) then
          SYMBOL_LIST.FIND_NEXT(TEMP_LIST);
        else
          while ((not STK_OF_LISTS.EMPTY(SEARCH_STACK)) and then
          (SYMBOL_LIST.LAST(TEMP_LIST))) loop
            STK_OF_LISTS.POP(SEARCH_STACK, TEMP_LIST);
            SYMBOL_LIST.RETRIEVE(TEMP_LIST, TEMP_POINTER);
            PRINT_RECORD(TEMP_POINTER.SYMBOL);
          end loop;
          exit when ((STK_OF_LISTS.EMPTY(SEARCH_STACK)) and then
                     (SYMBOL_LIST.LAST(TEMP_LIST)));
          SYMBOL_LIST.FIND_NEXT(TEMP_LIST);
        end if;
      end loop;
    end if;
    LAST_FOUND := null;
  end PRINT_SYMBOL_TABLE;

begin
  INITIALIZE_SYM_TAB;
end SYMBOL_TABLE;
```

APPENDIX F

"ADAFLOW" PROGRAM LISTING - CODE BLOCKER

```
--*************************************************************--
--                                                             --
--  TITLE:          ADAFLOW                                    --
--                                                             --
--  MODULE NAME:    PACKAGE CODE_BLOCKER                       --
--  FILE NAME:      BLOCKER.ADS                                --
--                                                             --
--  DATE CREATED:   31 MAR 88                                  --
--  LAST MODIFIED:  28 APR 88                                  --
--                                                             --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                   --
--                                                             --
--  DESCRIPTION:  This package defines the interface to the    --
--                CODE_BLOCKER module.                         --
--                                                             --
--*************************************************************--

with TOKEN_SCANNER; -- only for visibility of type SOURCE_RECORD

package CODE_BLOCKER is

  CODE_BLOCKER_UNDERFLOW : exception;
  CODE_BLOCKER_OVERFLOW  : exception;
  UNMATCHED_CODE_BLOCKS  : exception;

  procedure ENTER_CODE_BLOCK(SOURCE : in TOKEN_SCANNER.SOURCE_RECORD;
                             LABEL  : in string);
  -- post - A unique code block number, starting with the number 1 and
  --        continuing sequentially, is generated and associated with
  --        the new code block.  The current code block number is the
  --        new code block number.  The statement count is set to zero.

  procedure INCREMENT_STATEMENT_COUNT;
  -- pre  - A code block has been entered.
  -- post - Used to count the number of statements in a code
  --        block.  Initially zero, INCREMENT _STATEMENT_COUNT increases
  --        the count of statements encountered in the current
  --        code block by 1.
  -- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block has not been
  --                     entered.

  procedure DELETE_CODE_BLOCK_ENTER;
  -- pre  - A code block has been entered.
  -- post - The most recently entered code block is deleted and the state
  --        of the code blocker is restored to the state just prior to the
  --        erroneous code block entry.
  -- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block has not been
  --                     entered.
```

```
function  IS_CODE_BLOCK_ENTERED return boolean;
-- pre  - If a code block has been entered and not yet exited,
--        IS_CODE_BLOCK_ENTERED returns true, else returns false.

procedure EXIT_CODE_BLOCK(SOURCE :  in TOKEN_SCANNER.SOURCE_RECORD);
-- pre  - A code block has been entered.
-- post - The most recently entered code block is added to a list of
--        exited code blocks.  The next most recently entered code block,
--        if it exists, becomes the current code block.
-- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block has not been
--                     entered.

procedure REACTIVATE_CODE_BLOCK(CODE_BLOCK_NUMBER : in positive);
-- pre  - The code block number exists in the list of exited code blocks.
-- post - The code block is removed from the list of exited code blocks and
--        made the current code block.
-- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block does not exist
--                     in the list of exited code blocks with the named
--                     CODE_BLOCK_NUMBER.
--                     CODE_BLOCKER_UNDERFLOW if the block list is clear.

function  CURRENT_CODE_BLOCK_NUMBER return positive;
-- pre  - A code block has been entered and not yet exited.
-- post - CURRENT_CODE_BLOCK_NUMBER returns the number of the current,
--        code block that has most recently been entered.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the code blocker is
--                     not currently in a code block.

function  CURRENT_STATEMENT_COUNT return natural;
-- pre  - A code block has been entered.
-- post - CURRENT_STATEMENT_COUNT returns the count of
--        statements encountered in the current code block.
-- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block has not been
--                     entered.

procedure CLEAR_CODE_BLOCKER;
-- post - Clears the code blocker of all code blocks that have been entered
--        and of all code blocks in the list of exited code blocks.  The
--        current code block number is undefined.  The next code block
--        number to be generated is 1.

function  IS_CODE_BLOCK_LIST_CLEAR return boolean;
-- post - If no code blocks have been entered and exited then
--        IS_CODE_BLOCK_LIST_CLEAR returns true, else returns false.

function  IS_LAST_CODE_BLOCK return boolean;
-- pre  - The code block list is not clear.
-- post - If there are no other blocks of code in the list of code blocks,
--        IS_LAST_CODE_BLOCK returns true, else IS_LAST_CODE_BLOCK returns
--        false.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
```

```
procedure FIND_FIRST_CODE_BLOCK;
-- pre  - The code block list is not clear and no code blocks have been
--        entered and not yet exited.
-- post - Rewinds the code block list to the first block.  The current block
--        in the code block list is the first block in the code block list.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.

procedure FIND_NEXT_CODE_BLOCK;
-- pre  - The code block list is not at the last block and is not clear.
--        No code blocks have been entered and not yet exited.
-- post - The code blocker is advanced to the next block.  The current block
--        in the code block list is the next block in the code block list.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     CODE_BLOCK_OVERFLOW if at the last block in the list.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.

function  READ_CODE_BLOCK_NUMBER return positive;
-- pre  - The code block list is not clear.  No code blocks have been
--        entered and not yet exited.
-- post - READ_CODE_BLOCK_NUMBER returns the code block number of the
--        current code block in the code block list.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.

function  READ_CODE_BLOCK_STATEMENT_COUNT return natural;
-- pre  - The code block list is not clear.  No code blocks have been
--        entered and not yet exited.
-- post - READ_CODE_BLOCK_STATEMENT_COUNT returns the number of
--        statements recorded as encountered in the current code block
--        in the code block list.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.

function  READ_CODE_BLOCK_START return TOKEN_SCANNER.SOURCE_RECORD;
-- pre  - The code block list is not clear.  No code blocks have been
--        entered and not yet exited.
-- post - READ_CODE_BLOCK_START returns the record of origin of the
--        current code block in the code block list as it relates to the
--        source code.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.

function  READ_CODE_BLOCK_STOP return TOKEN_SCANNER.SOURCE_RECORD;
-- pre  - The code block list is not clear.  No code blocks have been
--        entered and not yet exited.
```

```
  -- post - READ_CODE_BLOCK_STOP returns the record of completion of the
  --        current code block in the code block list as it relates to the
  --        source code.
  -- exceptions raised - CODE_BLOCKER_UNDERFLOW if the code blocker is clear.
  --                     UNMATCHED_CODE_BLOCKS if a block has been entered
  --                     and not yet exited.

  function  READ_CODE_BLOCK_LABEL return string;
  -- pre  - The code block list is not clear.  No code blocks have been
  --        entered and not yet exited.
  -- post - READ_CODE_BLOCK_LABEL returns the label entered when the
  --        current code block in the code block list was entered.
  -- exceptions raised - CODE_BLOCKER_UNDERFLOW if the code blocker is clear.
  --                     UNMATCHED_CODE_BLOCKS if a block has been entered
  --                     and not yet exited.

end CODE_BLOCKER;
```

```
--*****************************************************************--
--                                                                 --
--  TITLE:          ADAFLOW                                        --
--                                                                 --
--  MODULE NAME:    PACKAGE CODE_BLOCKER                           --
--  FILE NAME:      BLOCKER.ADB                                    --
--                                                                 --
--  DATE CREATED:   31 MAR 88                                      --
--  LAST MODIFIED:  28 APR 88                                      --
--                                                                 --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                       --
--                                                                 --
--  DESCRIPTION:  This package implements the interface to the     --
--                CODE_BLOCKER module.                             --
--                                                                 --
--*****************************************************************--

with ORDERED_GENERIC_LIST,
     GENERIC_STACK,
     UNCHECKED_DEALLOCATION,
     TOKEN_SCANNER; -- only for visibility of type SOURCE_RECORD

package body CODE_BLOCKER is

  type CODE_BLOCK_RECORD is
    record
      BLOCK_NUMBER    : positive;
      STATEMENT_COUNT : natural := 0;
      START           : TOKEN_SCANNER.SOURCE_RECORD;
      STOP            : TOKEN_SCANNER.SOURCE_RECORD;
      LABEL           : string(1..TOKEN_SCANNER.LINESIZE) := (others => ' ');
      LABEL_LENGTH    : natural;
    end record;

  type CODE_BLOCK_POINTER is access CODE_BLOCK_RECORD;

  NEXT_BLOCK_NUMBER    : positive := 1;
  CURRENT_BLOCK_NUMBER : positive;

  package BLOCK_LIST  is new ORDERED_GENERIC_LIST(CODE_BLOCK_POINTER);
  package BLOCK_STACK is new GENERIC_STACK(CODE_BLOCK_POINTER);
  procedure FREE_CODE_BLOCK is new
             UNCHECKED_DEALLOCATION(CODE_BLOCK_RECORD, CODE_BLOCK_POINTER);

  BL : BLOCK_LIST.LIST;
  BS : BLOCK_STACK.STACK;

  procedure INITIALIZE_CODE_BLOCKER is
  SUCCESS : boolean;
  begin
    BLOCK_LIST.CREATE(BL, SUCCESS);
```

```
  if (not SUCCESS) then
    raise CODE_BLOCKER_OVERFLOW;
  end if;
  BLOCK_STACK.CREATE(BS, SUCCESS);
  if (not SUCCESS) then
    raise CODE_BLOCKER_OVERFLOW;
  end if;
  NEXT_BLOCK_NUMBER := 1;
end INITIALIZE_CODE_BLOCKER;

procedure ENTER_CODE_BLOCK(SOURCE : in TOKEN_SCANNER.SOURCE_RECORD;
                           LABEL : in string) is
-- post - A unique code block number, starting with the number 1 and
--        continuing sequentially, is generated and associated with
--        the new code block.  The current code block number is the
--        new code block number.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
  TEMP_POINTER := new CODE_BLOCK_RECORD;
  TEMP_POINTER.BLOCK_NUMBER := NEXT_BLOCK_NUMBER;
  CURRENT_BLOCK_NUMBER := NEXT_BLOCK_NUMBER;
  NEXT_BLOCK_NUMBER := NEXT_BLOCK_NUMBER + 1;
  TEMP_POINTER.STATEMENT_COUNT := 0;
  TEMP_POINTER.START := SOURCE;
  TEMP_POINTER.LABEL := (others => ' ');
  TEMP_POINTER.LABEL(1..LABEL'LAST) := LABEL;
  TEMP_POINTER.LABEL_LENGTH := LABEL'LENGTH;
  BLOCK_STACK.PUSH(BS, TEMP_POINTER);
end ENTER_CODE_BLOCK;

procedure INCREMENT_STATEMENT_COUNT is
-- pre  - A code block has been entered.
-- post - Used to count the number of statements in a code
--        block.  Initially zero, INCREMENT _STATEMENT_COUNT increases
--        the count of statements encountered in the current
--        code block by 1.
-- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block has not been
--                     entered.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
  if (BLOCK_STACK.EMPTY(BS)) then
    raise UNMATCHED_CODE_BLOCKS;
  else
    BLOCK_STACK.POP(BS, TEMP_POINTER);
    TEMP_POINTER.STATEMENT_COUNT :=
                              natural'SUCC(TEMP_POINTER.STATEMENT_COUNT);
    BLOCK_STACK.PUSH(BS, TEMP_POINTER);
  end if;
end INCREMENT_STATEMENT_COUNT;
```

```
procedure DELETE_CODE_BLOCK_ENTER is
-- pre  - A code block has been entered.
-- post - The most recently entered code block is deleted and the state
--        of the code blocker is restored to the state just prior to the
--        erroneous code block entry.
-- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block has not been
--                     entered.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
  if (BLOCK_STACK.EMPTY(BS)) then
    raise UNMATCHED_CODE_BLOCKS;
  else
    BLOCK_STACK.POP(BS, TEMP_POINTER);
    FREE_CODE_BLOCK(TEMP_POINTER);
    NEXT_BLOCK_NUMBER := NEXT_BLOCK_NUMBER - 1;
    if (not BLOCK_STACK.EMPTY(BS)) then
      BLOCK_STACK.TOP(BS, TEMP_POINTER);
      CURRENT_BLOCK_NUMBER := TEMP_POINTER.BLOCK_NUMBER;
    end if;
  end if;
end DELETE_CODE_BLOCK_ENTER;

function  IS_CODE_BLOCK_ENTERED return boolean is
-- pre  - If a code block has been entered and not yet exited,
--        IS_CODE_BLOCK_ENTERED returns true, else returns false.
begin
  return (not BLOCK_STACK.EMPTY(BS));
end IS_CODE_BLOCK_ENTERED;

procedure EXIT_CODE_BLOCK(SOURCE :  in TOKEN_SCANNER.SOURCE_RECORD) is
-- pre  - A code block has been entered.
-- post - The most recently entered code block is added to a list of
--        exited code blocks.  The next most recently entered code block,
--        if it exists, becomes the current code block.
-- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block has not been
--                     entered.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
  if (BLOCK_STACK.EMPTY(BS)) then
    raise UNMATCHED_CODE_BLOCKS;
  else
    BLOCK_STACK.POP(BS, TEMP_POINTER);
    TEMP_POINTER.STOP := SOURCE;
    BLOCK_LIST.INSERT(BL, TEMP_POINTER, TEMP_POINTER.BLOCK_NUMBER);
    if (not BLOCK_STACK.EMPTY(BS)) then
      BLOCK_STACK.TOP(BS, TEMP_POINTER);
      CURRENT_BLOCK_NUMBER := TEMP_POINTER.BLOCK_NUMBER;
    end if;
  end if;
end EXIT_CODE_BLOCK;
```

```
procedure REACTIVATE_CODE_BLOCK(CODE_BLOCK_NUMBER : in positive) is
-- pre  - The code block number exists in the list of exited code blocks.
-- post - The code block is removed from the list of exited code blocks and
--        made the current code block.
-- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block does not exist
--                     in the list of exited code blocks with the named
--                     CODE_BLOCK_NUMBER.
--                     CODE_BLOCKER_UNDERFLOW if the block list is clear.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
  if (BLOCK_LIST.EMPTY(BL)) then
    raise CODE_BLOCKER_UNDERFLOW;
  else
    BLOCK_LIST.FIND_FIRST(BL);
    BLOCK_LIST.RETRIEVE(BL, TEMP_POINTER);
    while (TEMP_POINTER.BLOCK_NUMBER /= CODE_BLOCK_NUMBER) loop
      if (BLOCK_LIST.LAST(BL)) then
        raise UNMATCHED_CODE_BLOCKS;
      else
        BLOCK_LIST.FIND_NEXT(BL);
        BLOCK_LIST.RETRIEVE(BL, TEMP_POINTER);
      end if;
    end loop;
    BLOCK_LIST.DELETE(BL);
    BLOCK_STACK.PUSH(BS, TEMP_POINTER);
    CURRENT_BLOCK_NUMBER := CODE_BLOCK_NUMBER;
  end if;
end REACTIVATE_CODE_BLOCK;

function  CURRENT_CODE_BLOCK_NUMBER return positive is
-- pre  - A code block has been entered and not yet exited.
-- post - CURRENT_CODE_BLOCK_NUMBER returns the number of the current,
--        code block that has most recently been entered.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the code blocker is
--                     not currently in a code block.
begin
  if (BLOCK_STACK.EMPTY(BS)) then
    raise CODE_BLOCKER_UNDERFLOW;
  else
    return (CURRENT_BLOCK_NUMBER);
  end if;
end CURRENT_CODE_BLOCK_NUMBER;

function  CURRENT_STATEMENT_COUNT return natural is
-- pre  - A code block has been entered.
-- post - CURRENT_STATEMENT_COUNT returns the count of
--        statements encountered in the current code block.
-- exceptions raised - UNMATCHED_CODE_BLOCKS if a code block has not been
--                     entered.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
```

```
  if (BLOCK_STACK.EMPTY(BS)) then
    raise UNMATCHED_CODE_BLOCKS;
  else
    BLOCK_STACK.TOP(BS, TEMP_POINTER);
    return (TEMP_POINTER.STATEMENT_COUNT);
  end if;
end CURRENT_STATEMENT_COUNT;

procedure CLEAR_CODE_BLOCKER is
-- post - Clears the code blocker of all code blocks that have been entered
--        and of all code blocks in the list of exited code blocks.  The
--        current code block number is undefined.  The next code block
--        number to be generated is 1.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
  while (not BLOCK_LIST.EMPTY(BL)) loop
    BLOCK_LIST.RETRIEVE(BL, TEMP_POINTER);
    FREE_CODE_BLOCK(TEMP_POINTER);
    BLOCK_LIST.DELETE(BL);
  end loop;
  while (not BLOCK_STACK.EMPTY(BS)) loop
    BLOCK_STACK.POP(BS, TEMP_POINTER);
    FREE_CODE_BLOCK(TEMP_POINTER);
  end loop;
  NEXT_BLOCK_NUMBER := 1;
end CLEAR_CODE_BLOCKER;

function  IS_CODE_BLOCK_LIST_CLEAR return boolean is
-- post - If no code blocks have been both entered and exited then
--        IS_CODE_BLOCK_LIST_CLEAR returns true, else returns false.
begin
  return (BLOCK_LIST.EMPTY(BL));
end IS_CODE_BLOCK_LIST_CLEAR;

function  IS_LAST_CODE_BLOCK return boolean is
-- pre  - The code block list is not clear.
-- post - If there are no other blocks of code in the list of code blocks,
--        IS_LAST_CODE_BLOCK returns true, else IS_LAST_CODE_BLOCK returns
--        false.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
begin
  if (BLOCK_LIST.EMPTY(BL)) then
    raise CODE_BLOCKER_UNDERFLOW;
  else
    return (BLOCK_LIST.LAST(BL));
  end if;
end IS_LAST_CODE_BLOCK;

procedure FIND_FIRST_CODE_BLOCK is
-- pre  - The code block list is not clear and no code blocks have been
--        entered and not yet exited.
```

```
-- post - Rewinds the code block list to the first block.  The current block
--        in the code block list is the first block in the code block list.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.
begin
  if (BLOCK_LIST.EMPTY(BL)) then
    raise CODE_BLOCKER_UNDERFLOW;
  elsif (not BLOCK_STACK.EMPTY(BS)) then
    raise UNMATCHED_CODE_BLOCKS;
  else
    BLOCK_LIST.FIND_FIRST(BL);
  end if;
end FIND_FIRST_CODE_BLOCK;

procedure FIND_NEXT_CODE_BLOCK is
-- pre  - The code block list is not at the last block and is not clear.
--        No code blocks have been entered and not yet exited.
-- post - The code blocker is advanced to the next block.  The current block
--        in the code block list is the next block in the code block list.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     CODE_BLOCK_OVERFLOW if at the last block in the list.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.
begin
  if (BLOCK_LIST.EMPTY(BL)) then
    raise CODE_BLOCKER_UNDERFLOW;
  elsif (BLOCK_LIST.LAST(BL)) then
    raise CODE_BLOCKER_OVERFLOW;
  elsif (not BLOCK_STACK.EMPTY(BS)) then
    raise UNMATCHED_CODE_BLOCKS;
  else
    BLOCK_LIST.FIND_NEXT(BL);
  end if;
end FIND_NEXT_CODE_BLOCK;

function  READ_CODE_BLOCK_NUMBER return positive is
-- pre  - The code block list is not clear.  No code blocks have been
--        entered and not yet exited.
-- post - READ_CODE_BLOCK_NUMBER returns the code block number of the
--        current code block in the code block list.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
  if (BLOCK_LIST.EMPTY(BL)) then
    raise CODE_BLOCKER_UNDERFLOW;
  elsif (not BLOCK_STACK.EMPTY(BS)) then
    raise UNMATCHED_CODE_BLOCKS;
  else
```

```
    BLOCK_LIST.RETRIEVE(BL, TEMP_POINTER);
    return (TEMP_POINTER.BLOCK_NUMBER);
  end if;
end READ_CODE_BLOCK_NUMBER;

function  READ_CODE_BLOCK_STATEMENT_COUNT return natural is
-- pre  - The code block list is not clear.  No code blocks have been
--        entered and not yet exited.
-- post - READ_CODE_BLOCK_STATEMENT_COUNT returns the number of
--        statements recorded as encountered in the current code block
--        in the code block list.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
  if (BLOCK_LIST.EMPTY(BL)) then
    raise CODE_BLOCKER_UNDERFLOW;
  elsif (not BLOCK_STACK.EMPTY(BS)) then
    raise UNMATCHED_CODE_BLOCKS;
  else
    BLOCK_LIST.RETRIEVE(BL, TEMP_POINTER);
    return (TEMP_POINTER.STATEMENT_COUNT);
  end if;
end READ_CODE_BLOCK_STATEMENT_COUNT;

function  READ_CODE_BLOCK_START return TOKEN_SCANNER.SOURCE_RECORD is
-- pre  - The code block list is not clear.  No code blocks have been
--        entered and not yet exited.
-- post - READ_CODE_BLOCK_START returns the record of origin of the
--        current code block in the code block list as it relates to the
--        source code.
-- exceptions raised - CODE_BLOCKER_UNDERFLOW if the block list is clear.
--                     UNMATCHED_CODE_BLOCKS if a block has been entered
--                     and not yet exited.
TEMP_POINTER : CODE_BLOCK_POINTER;
begin
  if (BLOCK_LIST.EMPTY(BL)) then
    raise CODE_BLOCKER_UNDERFLOW;
  elsif (not BLOCK_STACK.EMPTY(BS)) then
    raise UNMATCHED_CODE_BLOCKS;
  else
    BLOCK_LIST.RETRIEVE(BL, TEMP_POINTER);
    return (TEMP_POINTER.START);
  end if;
end READ_CODE_BLOCK_START;

function  READ_CODE_BLOCK_STOP return TOKEN_SCANNER.SOURCE_RECORD is
-- pre  - The code block list is not clear.  No code blocks have been
--        entered and not yet exited.
-- post - READ_CODE_BLOCK_STOP returns the record of completion of the
```

```
  --        current code block in the code block list as it relates to the
  --        source code.
  -- exceptions raised - CODE_BLOCKER_UNDERFLOW if the code blocker is clear.
  --                     UNMATCHED_CODE_BLOCKS if a block has been entered
  --                     and not yet exited.
  TEMP_POINTER : CODE_BLOCK_POINTER;
  begin
    if (BLOCK_LIST.EMPTY(BL)) then
      raise CODE_BLOCKER_UNDERFLOW;
    elsif (not BLOCK_STACK.EMPTY(BS)) then
      raise UNMATCHED_CODE_BLOCKS;
    else
      BLOCK_LIST.RETRIEVE(BL, TEMP_POINTER);
      return (TEMP_POINTER.STOP);
    end if;
  end READ_CODE_BLOCK_STOP;

  function  READ_CODE_BLOCK_LABEL return string is
  -- pre  - The code block list is not clear.  No code blocks have been
  --        entered and not yet exited.
  -- post - READ_CODE_BLOCK_LABEL returns the label entered when the
  --        current code block in the code block list was entered.
  -- exceptions raised - CODE_BLOCKER_UNDERFLOW if the code blocker is clear.
  --                     UNMATCHED_CODE_BLOCKS if a block has been entered
  --                     and not yet exited.
  TEMP_POINTER : CODE_BLOCK_POINTER;
  begin
    if (BLOCK_LIST.EMPTY(BL)) then
      raise CODE_BLOCKER_UNDERFLOW;
    elsif (not BLOCK_STACK.EMPTY(BS)) then
      raise UNMATCHED_CODE_BLOCKS;
    else
      BLOCK_LIST.RETRIEVE(BL, TEMP_POINTER);
      return (TEMP_POINTER.LABEL(1..TEMP_POINTER.LABEL_LENGTH));
    end if;
  end READ_CODE_BLOCK_LABEL;

begin
  INITIALIZE_CODE_BLOCKER;
end CODE_BLOCKER;
```

APPENDIX G

"ADAFLOW" PROGRAM LISTING - TOKEN MATCHER

```
--*****************************************************************--
--                                                                  --
--  TITLE:          ADAFLOW                                         --
--                                                                  --
--  MODULE NAME:    PACKAGE TOKEN_MATCHER                           --
--  FILE NAME:      MATCH.ADS                                       --
--                                                                  --
--  DATE CREATED:   18 FEB 88                                       --
--  LAST MODIFIED:  28 APR 88                                       --
--                                                                  --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                        --
--                                                                  --
--  DESCRIPTION:  This package defines the interface to the        --
--                module that identifies each individual           --
--                token and manages the TOKEN_SCANNER.  The        --
--                TOKEN_MATCHER is the sole manager of the         --
--                TOKEN_SCANNER interface and all access to the    --
--                TOKEN_SCANNER interface is through TOKEN_        --
--                MATCHER.  This restriction does not apply to     --
--                types specified in the TOKEN SCANNER             --
--                interface.  Types specified in the TOKEN_        --
--                SCANNER interface are available for global use.--
--                                                                  --
--*****************************************************************--

with TOKEN_SCANNER;

package TOKEN_MATCHER is

-- The following token codes define the terminals of the ADA language.

-- basic tokens
  TOKEN_IDENTIFIER            : constant integer := 1;
  TOKEN_NUMERIC_LITERAL       : constant integer := 2;
  TOKEN_CHARACTER_LITERAL     : constant integer := 3;
  TOKEN_STRING_LITERAL        : constant integer := 4;
-- reserved word tokens
  TOKEN_END                   : constant integer := 5;
  TOKEN_BEGIN                 : constant integer := 6;
  TOKEN_IF                    : constant integer := 7;
  TOKEN_THEN                  : constant integer := 8;
  TOKEN_ELSIF                 : constant integer := 9;
  TOKEN_ELSE                  : constant integer := 10;
```

```
TOKEN_WHILE              : constant integer := 11;
TOKEN_LOOP               : constant integer := 12;
TOKEN_CASE               : constant integer := 13;
TOKEN_WHEN               : constant integer := 14;
TOKEN_DECLARE            : constant integer := 15;
TOKEN_FOR                : constant integer := 16;
TOKEN_OTHERS             : constant integer := 17;
TOKEN_RETURN             : constant integer := 18;
TOKEN_EXIT               : constant integer := 19;
TOKEN_PROCEDURE          : constant integer := 20;
TOKEN_FUNCTION           : constant integer := 21;
TOKEN_WITH               : constant integer := 22;
TOKEN_USE                : constant integer := 23;
TOKEN_PACKAGE            : constant integer := 24;
TOKEN_BODY               : constant integer := 25;
TOKEN_RANGE              : constant integer := 26;
TOKEN_IN                 : constant integer := 27;
TOKEN_OUT                : constant integer := 28;
TOKEN_SUBTYPE            : constant integer := 29;
TOKEN_TYPE               : constant integer := 30;
TOKEN_IS                 : constant integer := 31;
TOKEN_NULL               : constant integer := 32;
TOKEN_ACCESS             : constant integer := 33;
TOKEN_ARRAY              : constant integer := 34;
TOKEN_DIGITS             : constant integer := 35;
TOKEN_DELTA              : constant integer := 36;
TOKEN_RECORD_STRUCTURE   : constant integer := 37;
TOKEN_CONSTANT           : constant integer := 38;
TOKEN_NEW                : constant integer := 39;
TOKEN_EXCEPTION          : constant integer := 40;
TOKEN_RENAMES            : constant integer := 41;
TOKEN_PRIVATE            : constant integer := 42;
TOKEN_LIMITED            : constant integer := 43;
TOKEN_TASK               : constant integer := 44;
TOKEN_ENTRY              : constant integer := 45;
TOKEN_ACCEPT             : constant integer := 46;
TOKEN_DELAY              : constant integer := 47;
TOKEN_SELECT             : constant integer := 48;
TOKEN_TERMINATE          : constant integer := 49;
TOKEN_ABORT              : constant integer := 50;
TOKEN_SEPARATE           : constant integer := 51;
TOKEN_RAISE              : constant integer := 52;
TOKEN_GENERIC            : constant integer := 53;
TOKEN_AT                 : constant integer := 54;
TOKEN_REVERSE            : constant integer := 55;
TOKEN_DO                 : constant integer := 56;
TOKEN_GOTO               : constant integer := 57;
TOKEN_OF                 : constant integer := 58;
TOKEN_ALL                : constant integer := 59;
TOKEN_PRAGMA             : constant integer := 60;
TOKEN_AND                : constant integer := 61;
```

```
  TOKEN_OR                    : constant integer := 62;
  TOKEN_NOT                   : constant integer := 63;
  TOKEN_XOR                   : constant integer := 64;
  TOKEN_MOD                   : constant integer := 65;
  TOKEN_REM                   : constant integer := 66;
  TOKEN_ABSOLUTE              : constant integer := 67;
-- delimiter tokens
  TOKEN_ASTERISK              : constant integer := 68;
  TOKEN_SLASH                 : constant integer := 69;
  TOKEN_EXPONENT              : constant integer := 70;
  TOKEN_PLUS                  : constant integer := 71;
  TOKEN_MINUS                 : constant integer := 72;
  TOKEN_AMPERSAND             : constant integer := 73;
  TOKEN_EQUALS                : constant integer := 74;
  TOKEN_NOT_EQUALS            : constant integer := 75;
  TOKEN_LESS_THAN             : constant integer := 76;
  TOKEN_LESS_THAN_EQUALS      : constant integer := 77;
  TOKEN_GREATER_THAN          : constant integer := 78;
  TOKEN_GREATER_THAN_EQUALS   : constant integer := 79;
  TOKEN_ASSIGNMENT            : constant integer := 80;
  TOKEN_SEMICOLON             : constant integer := 81;
  TOKEN_PERIOD                : constant integer := 82;
  TOKEN_LEFT_PAREN            : constant integer := 83;
  TOKEN_RIGHT_PAREN           : constant integer := 84;
  TOKEN_COLON                 : constant integer := 85;
  TOKEN_COMMA                 : constant integer := 86;
  TOKEN_APOSTROPHE            : constant integer := 87;
  TOKEN_RANGE_DOTS            : constant integer := 88;
  TOKEN_ARROW                 : constant integer := 89;
  TOKEN_BAR                   : constant integer := 90;
  TOKEN_BRACKETS              : constant integer := 91;
  TOKEN_LEFT_BRACKET          : constant integer := 92;
  TOKEN_RIGHT_BRACKET         : constant integer := 93;

  procedure SET_UP_TOKEN_MATCHER(FILE_NAME : string);
  -- pre  - must be called before any of the defined interfaces in
  --        TOKEN_MATCHER are invoked.  Any previously set up FILE_NAME
  --        must be released by RELEASE_TOKEN_SCANNER.
  -- post - the TOKEN_MATCHER interfaces are defined.

  procedure RELEASE_TOKEN_MATCHER;
  -- pre  - TOKEN_MATCHER has been set up.
  -- post - all TOKEN_MATCHER interfaces are undefined with the
  --        exception of SET_UP_TOKEN_MATCHER.
  --        TOKEN_MATCHER may be set up for another FILE_NAME.  The
  --        TOKEN_MATCHER must be released prior to main program
  --        termination.

  function MATCH(TOKEN_CODE : in positive) return boolean;
  -- pre  - TOKEN_MATCHER has been set up.
  -- post - if the current token under the read head of the TOKEN_SCANNER
```

```
--         matches the TOKEN_COOE then MATCH is TRUE and the read head of
--         the TOKEN_SCANNER is advanced one token. Else MATCH is FALSE
--         and the read head of the TOKEN_SCANNER does not advance.

  procedure MATCHEO_TOKEN(TOKEN : out TOKEN_SCANNER.TOKEN_RECORD_TYPE);
  -- pre  - TOKEN_MATCHER has been set up and at least one call to the
  --        function MATCH has returned TRUE.
  -- post - TOKEN contains the token that caused the last call to MATCH
  --        to be TRUE.  NOTE - All identifiers are converted to upper
  --        case by the token matcher and all reserved words are converted
  --        to lower case by the token matcher regardless of original format
  --        in the source code.  All other token types are left in original
  --        source code format.

  procedure CURRENT_TOKEN(TOKEN : out TOKEN_SCANNER.TOKEN_RECORO_TYPE);
  -- pre  - TOKEN_MATCHER has been set up.
  -- post - TOKEN contains the token that is under the TOKEN_SCANNER's
  --        read head.

  procedure NEXT_TOKEN(TOKEN : out TOKEN_SCANNER.TOKEN_RECORO_TYPE);
  -- pre  - TOKEN_MATCHER has been set up.
  -- post - TOKEN contains the token that is next to be read by the
  --        TOKEN_SCANNERS read head.

  function LINES_CHECKED return positive;
  -- pre  - TOKEN_MATCHER has been set up.
  -- post - returns the number of lines of code that have been checked
  --        by the TOKEN_MATCHER.

  function VALIO_COMMENTS return natural;
  -- pre  - TOKEN_MATCHER has been set up.
  -- post - returns the number of "meaningful" comments seen by the
  --        TOKEN_MATCHER.  A "meaningful" comment is defined as a comment
  --        that contains at least one letter or digit.

end TOKEN_MATCHER;
```

```
--*****************************************************************--
--                                                                 --
--  TITLE:          ADAFLOW                                        --
--                                                                 --
--  MODULE NAME:    PACKAGE TOKEN_MATCHER                          --
--  FILE NAME:      MATCH.ADB                                      --
--                                                                 --
--  DATE CREATED:   18 FEB 88                                      --
--  LAST MODIFIED:  28 APR 88                                      --
--                                                                 --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                       --
--                                                                 --
--  DESCRIPTION:  This package implements the interface to the     --
--                module that identifies each individual           --
--                token and manages the TOKEN_SCANNER.  The        --
--                TOKEN_MATCHER is the sole manager of the         --
--                TOKEN_SCANNER interface and all access to the    --
--                TOKEN_SCANNER interface is through TOKEN_        --
--                MATCHER.  This restriction does not apply to     --
--                types specified in the TOKEN SCANNER             --
--                interface.  Types specified in the TOKEN_        --
--                SCANNER interface are available for global use.--
--                                                                 --
--*****************************************************************--

with TOKEN_SCANNER, TEXT_IO;

package body TOKEN_MATCHER is

  SOURCE_FILE         : TEXT_IO.file_type;
  HOLD_TOKEN          : TOKEN_SCANNER.TOKEN_RECORD_TYPE;

  procedure SET_UP_TOKEN_MATCHER(FILE_NAME : string) is
  -- pre  - must be called before any of the defined interfaces in
  --        TOKEN_MATCHER are invoked.  Any previously set up FILE_NAME
  --        must be released by RELEASE_TOKEN_SCANNER.
  -- post - the TOKEN_MATCHER interfaces are defined.
  begin
    TEXT_IO.open(SOURCE_FILE, TEXT_IO.in_file, FILE_NAME, "");
    TEXT_IO.reset(SOURCE_FILE);
    TOKEN_SCANNER.SET_UP_TOKEN_SCANNER(SOURCE_FILE);
  end SET_UP_TOKEN_MATCHER;

  procedure RELEASE_TOKEN_MATCHER is
  -- pre  - TOKEN_MATCHER has been set up.
  -- post - all TOKEN_MATCHER interfaces are undefined with the
  --        exception of SET_UP_TOKEN_MATCHER.
  --        TOKEN_MATCHER may be set up for another FILE_NAME.  the
  --        TOKEN_MATCHER must be released prior to main program
  --        termination.
  begin
```

```
  TOKEN_SCANNER.RELEASE_TOKEN_SCANNER(SOURCE_FILE);
end RELEASE_TOKEN_MATCHER;

function MATCH(TOKEN_CODE : in positive) return boolean is
-- pre  - TOKEN_MATCHER has been set up.
-- post - if the current token under the read head of the TOKEN_SCANNER
--        matches the TOKEN_CODE then MATCH is true and the read head of
--        the TOKEN_SCANNER is advanced one token.  Else MATCH is false
--        and the read head of the TOKEN_SCANNER does not advance.
use TOKEN_SCANNER;
subtype BASIC_TOKENS is
        positive range TOKEN_IDENTIFIER..TOKEN_STRING_LITERAL;
subtype RESERVED_TOKENS is
        positive range TOKEN_END..TOKEN_ABSOLUTE;
subtype DELIMITER_TOKENS is
        positive range TOKEN_ASTERISK..TOKEN_RIGHT_BRACKET;
CURRENT_TOKEN     : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
TEST_TOKEN        : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
IS_SAME           : boolean := FALSE;

  function ASSIGN(TEST_STRING : in string) return
                TOKEN_SCANNER.TOKEN_RECORD_TYPE is
  TEMP_TOKEN : TOKEN_SCANNER.TOKEN_RECORD_TYPE;
  begin
    TEMP_TOKEN.LEXEME_SIZE := TEST_STRING'LENGTH;
    TEMP_TOKEN.LEXEME      := (others => ' ');
    TEMP_TOKEN.LEXEME(1..TEST_STRING'LAST) := TEST_STRING;
    TEMP_TOKEN.SOURCE := CURRENT_TOKEN.SOURCE;
    if (TOKEN_CODE in RESERVED_TOKENS) then
      TEMP_TOKEN.TOKEN_TYPE := TOKEN_SCANNER.RESERVED_WORD;
    elsif (TOKEN_CODE in DELIMITER_TOKENS) then
      TEMP_TOKEN.TOKEN_TYPE := TOKEN_SCANNER.DELIMITER;
    end if;
    return TEMP_TOKEN;
  end ASSIGN;

  procedure CONVERT_UPPER_CASE(TOKEN :
                          in out TOKEN_SCANNER.TOKEN_RECORD_TYPE) is
  subtype UPPER_CASE_LETTER is character range 'A'..'Z';
  subtype LOWER_CASE_LETTER is character range 'a'..'z';
  begin
    for LEXEME_INDEX in 1..TOKEN.LEXEME_SIZE loop
      if TOKEN.LEXEME(LEXEME_INDEX) in LOWER_CASE_LETTER then
        TOKEN.LEXEME(LEXEME_INDEX) :=
          UPPER_CASE_LETTER'VAL(LOWER_CASE_LETTER'POS(
          TOKEN.LEXEME(LEXEME_INDEX)) - 32);
      end if;
    end loop;
  end CONVERT_UPPER_CASE;
```

```
  procedure CONVERT_LOWER_CASE(TOKEN :
                             in out TOKEN_SCANNER.TOKEN_RECORD_TYPE) is
  subtype UPPER_CASE_LETTER is character range 'A'..'Z';
  subtype LOWER_CASE_LETTER is character range 'a'..'z';
  begin
    for LEXEME_INDEX in 1..TOKEN.LEXEME_SIZE loop
      if TOKEN.LEXEME(LEXEME_INDEX) in UPPER_CASE_LETTER then
        TOKEN.LEXEME(LEXEME_INDEX) :=
          LOWER_CASE_LETTER'VAL(UPPER_CASE_LETTER'POS(
          TOKEN.LEXEME(LEXEME_INDEX)) + 32);
      end if;
    end loop;
  end CONVERT_LOWER_CASE;
begin
  TOKEN_SCANNER.LOOK_TOKEN(SOURCE_FILE, CURRENT_TOKEN);
  if (TOKEN_CODE in BASIC_TOKENS) then
    case TOKEN_CODE is
      when TOKEN_IDENTIFIER =>
        IS_SAME := (CURRENT_TOKEN.TOKEN_TYPE = TOKEN_SCANNER.IDENTIFIER);
        if (IS_SAME) then
          CONVERT_UPPER_CASE(CURRENT_TOKEN);
        end if;
      when TOKEN_NUMERIC_LITERAL =>
        IS_SAME := (CURRENT_TOKEN.TOKEN_TYPE = TOKEN_SCANNER.NUMERIC_LIT);
      when TOKEN_CHARACTER_LITERAL =>
        IS_SAME := (CURRENT_TOKEN.TOKEN_TYPE = TOKEN_SCANNER.CHARACTER_LIT);
      when TOKEN_STRING_LITERAL =>
        IS_SAME := (CURRENT_TOKEN.TOKEN_TYPE = TOKEN_SCANNER.STRING_LIT);
      when others => null;
    end case;
  else
    CONVERT_LOWER_CASE(CURRENT_TOKEN);
    case TOKEN_CODE is
      when TOKEN_END =>
        TEST_TOKEN := ASSIGN("end");
      when TOKEN_BEGIN =>
        TEST_TOKEN := ASSIGN("begin");
      when TOKEN_IF =>
        TEST_TOKEN := ASSIGN("if");
      when TOKEN_THEN =>
        TEST_TOKEN := ASSIGN("then");
      when TOKEN_ELSIF =>
        TEST_TOKEN := ASSIGN("elsif");
      when TOKEN_ELSE =>
        TEST_TOKEN := ASSIGN("else");
      when TOKEN_WHILE =>
        TEST_TOKEN := ASSIGN("while");
      when TOKEN_LOOP =>
        TEST_TOKEN := ASSIGN("loop");
      when TOKEN_CASE =>
        TEST_TOKEN := ASSIGN("case");
```

```
when TOKEN_WHEN =>
  TEST_TOKEN := ASSIGN("when");
when TOKEN_DECLARE =>
  TEST_TOKEN := ASSIGN("declare");
when TOKEN_FOR =>
  TEST_TOKEN := ASSIGN("for");
when TOKEN_OTHERS =>
  TEST_TOKEN := ASSIGN("others");
when TOKEN_RETURN =>
  TEST_TOKEN := ASSIGN("return");
when TOKEN_EXIT =>
  TEST_TOKEN := ASSIGN("exit");
when TOKEN_PROCEDURE =>
  TEST_TOKEN := ASSIGN("procedure");
when TOKEN_FUNCTION =>
  TEST_TOKEN := ASSIGN("function");
when TOKEN_WITH =>
  TEST_TOKEN := ASSIGN("with");
when TOKEN_USE =>
  TEST_TOKEN := ASSIGN("use");
when TOKEN_PACKAGE =>
  TEST_TOKEN := ASSIGN("package");
when TOKEN_BODY =>
  TEST_TOKEN := ASSIGN("body");
when TOKEN_RANGE =>
  TEST_TOKEN := ASSIGN("range");
when TOKEN_IN =>
  TEST_TOKEN := ASSIGN("in");
when TOKEN_OUT =>
  TEST_TOKEN := ASSIGN("out");
when TOKEN_SUBTYPE =>
  TEST_TOKEN := ASSIGN("subtype");
when TOKEN_TYPE =>
  TEST_TOKEN := ASSIGN("type");
when TOKEN_IS =>
  TEST_TOKEN := ASSIGN("is");
when TOKEN_NULL =>
  TEST_TOKEN := ASSIGN("null");
when TOKEN_ACCESS =>
  TEST_TOKEN := ASSIGN("access");
when TOKEN_ARRAY =>
  TEST_TOKEN := ASSIGN("array");
when TOKEN_DIGITS =>
  TEST_TOKEN := ASSIGN("digits");
when TOKEN_DELTA =>
  TEST_TOKEN := ASSIGN("delta");
when TOKEN_RECORD_STRUCTURE =>
  TEST_TOKEN := ASSIGN("record");
when TOKEN_CONSTANT =>
  TEST_TOKEN := ASSIGN("constant");
when TOKEN_NEW =>
```

```
  TEST_TOKEN := ASSIGN("new");
when TOKEN_EXCEPTION =>
  TEST_TOKEN := ASSIGN("exception");
when TOKEN_RENAMES =>
  TEST_TOKEN := ASSIGN("renames");
when TOKEN_PRIVATE =>
  TEST_TOKEN := ASSIGN("private");
when TOKEN_LIMITED =>
  TEST_TOKEN := ASSIGN("limited");
when TOKEN_TASK =>
  TEST_TOKEN := ASSIGN("task");
when TOKEN_ENTRY =>
  TEST_TOKEN := ASSIGN("entry");
when TOKEN_ACCEPT =>
  TEST_TOKEN := ASSIGN("accept");
when TOKEN_DELAY =>
  TEST_TOKEN := ASSIGN("delay");
when TOKEN_SELECT =>
  TEST_TOKEN := ASSIGN("select");
when TOKEN_TERMINATE =>
  TEST_TOKEN := ASSIGN("terminate");
when TOKEN_ABORT =>
  TEST_TOKEN := ASSIGN("abort");
when TOKEN_SEPARATE =>
  TEST_TOKEN := ASSIGN("separate");
when TOKEN_RAISE =>
  TEST_TOKEN := ASSIGN("raise");
when TOKEN_GENERIC =>
  TEST_TOKEN := ASSIGN("generic");
when TOKEN_AT =>
  TEST_TOKEN := ASSIGN("at");
when TOKEN_REVERSE =>
  TEST_TOKEN := ASSIGN("reverse");
when TOKEN_DO =>
  TEST_TOKEN := ASSIGN("do");
when TOKEN_GOTO =>
  TEST_TOKEN := ASSIGN("goto");
when TOKEN_OF =>
  TEST_TOKEN := ASSIGN("of");
when TOKEN_ALL =>
  TEST_TOKEN := ASSIGN("all");
when TOKEN_PRAGMA =>
  TEST_TOKEN := ASSIGN("pragma");
when TOKEN_AND =>
  TEST_TOKEN := ASSIGN("and");
when TOKEN_OR =>
  TEST_TOKEN := ASSIGN("or");
when TOKEN_NOT =>
  TEST_TOKEN := ASSIGN("not");
when TOKEN_XOR =>
  TEST_TOKEN := ASSIGN("xor");
```

```
when TOKEN_MOD =>
  TEST_TOKEN := ASSIGN("mod");
when TOKEN_REM =>
  TEST_TOKEN := ASSIGN("rem");
when TOKEN_ABSOLUTE =>
  TEST_TOKEN := ASSIGN("abs");
when TOKEN_ASTERISK =>
  TEST_TOKEN := ASSIGN("*");
when TOKEN_SLASH =>
  TEST_TOKEN := ASSIGN("/");
when TOKEN_EXPONENT =>
  TEST_TOKEN := ASSIGN("**");
when TOKEN_PLUS =>
  TEST_TOKEN := ASSIGN("+");
when TOKEN_MINUS =>
  TEST_TOKEN := ASSIGN("-");
when TOKEN_AMPERSAND =>
  TEST_TOKEN := ASSIGN("&");
when TOKEN_EQUALS =>
  TEST_TOKEN := ASSIGN("=");
when TOKEN_NOT_EQUALS =>
  TEST_TOKEN := ASSIGN("/=");
when TOKEN_LESS_THAN =>
  TEST_TOKEN := ASSIGN("<");
when TOKEN_LESS_THAN_EQUALS =>
  TEST_TOKEN := ASSIGN("<=");
when TOKEN_GREATER_THAN =>
  TEST_TOKEN := ASSIGN(">");
when TOKEN_GREATER_THAN_EQUALS =>
  TEST_TOKEN := ASSIGN(">=");
when TOKEN_ASSIGNMENT =>
  TEST_TOKEN := ASSIGN(":=");
when TOKEN_COMMA =>
  TEST_TOKEN := ASSIGN(",");
when TOKEN_SEMICOLON =>
  TEST_TOKEN := ASSIGN(";");
when TOKEN_PERIOD =>
  TEST_TOKEN := ASSIGN(".");
when TOKEN_LEFT_PAREN =>
  TEST_TOKEN := ASSIGN("(");
when TOKEN_RIGHT_PAREN =>
  TEST_TOKEN := ASSIGN(")");
when TOKEN_COLON =>
  TEST_TOKEN := ASSIGN(":");
when TOKEN_APOSTROPHE =>
  TEST_TOKEN := ASSIGN("'");
when TOKEN_RANGE_DOTS =>
  TEST_TOKEN := ASSIGN("..");
when TOKEN_ARROW =>
  TEST_TOKEN := ASSIGN("=>");
when TOKEN_BAR =>
```

```
        TEST_TOKEN := ASSIGN("|");
      when TOKEN_BRACKETS =>
        TEST_TOKEN := ASSIGN("<>");
      when TOKEN_LEFT_BRACKET =>
        TEST_TOKEN := ASSIGN("<<");
      when TOKEN_RIGHT_BRACKET =>
        TEST_TOKEN := ASSIGN(">>");
      when others => null;
    end case;
    IS_SAME := (CURRENT_TOKEN = TEST_TOKEN);
  end if;
  if (IS_SAME) then
    HOLD_TOKEN := CURRENT_TOKEN;
    TOKEN_SCANNER.CONSUME_TOKEN(SOURCE_FILE);
  end if;
  return (IS_SAME);
end MATCH;

procedure MATCHED_TOKEN(TOKEN : out TOKEN_SCANNER.TOKEN_RECORD_TYPE) is
-- pre  - TOKEN_MATCHER has been set up and at least one call to the
--        function MATCH has returned TRUE;
-- post - TOKEN contains the token that caused the last call to MATCH
--        to be TRUE.  NOTE - All identifiers are converted to upper case
--        by the token matcher and all reserved words are converted to lower
--        case by the token matcher regardless of the format in the source
--        code.  All other token types are uneffected by the token matcher.
begin
  TOKEN := HOLD_TOKEN;
end MATCHED_TOKEN;

procedure CURRENT_TOKEN(TOKEN : out TOKEN_SCANNER.TOKEN_RECORD_TYPE) is
-- pre  - TOKEN_MATCHER has been set up.
-- post - TOKEN contains the token that is under the TOKEN_SCANNER's
--        read head.
begin
  TOKEN_SCANNER.LOOK_TOKEN(SOURCE_FILE, TOKEN);
end CURRENT_TOKEN;

procedure NEXT_TOKEN(TOKEN : out TOKEN_SCANNER.TOKEN_RECORD_TYPE) is
-- pre  - TOKEN_MATCHER has been set up.
-- post - TOKEN contains the token that is next to be read by the
--        TOKEN_SCANNERS read head.
begin
  TOKEN_SCANNER.LOOK_AHEAD_TOKEN(SOURCE_FILE, TOKEN);
end NEXT_TOKEN;

function LINES_CHECKED return positive is
-- pre  - TOKEN_MATCHER has been set up.
-- post - returns the number of lines of code that have been checked
--        by the TOKEN_MATCHER.
begin
```

```
    return (TOKEN_SCANNER.LINES_SCANNED(SOURCE_FILE));
  end LINES_CHECKED;

  function VALID_COMMENTS return natural is
  -- pre  - TOKEN_MATCHER has been set up.
  -- post - returns the number of "meaningful" comments seen by the
  --        TOKEN_MATCHER.  A "meaningful" comment is defined as a comment
  --        that contains at least one letter or digit.
  begin
    return (TOKEN_SCANNER.COMMENTS_SCANNED(SOURCE_FILE));
  end VALID_COMMENTS;

end TOKEN_MATCHER;
```

APPENDIX H

"ADAFLOW" PROGRAM LISTING - TOKEN SCANNER

```
--***************************************************************--
--                                                               --
--  TITLE:          ADAFLOW                                      --
--                                                               --
--  MODULE NAME:    PACKAGE TOKEN_SCANNER                        --
--  FILE NAME:      TOKEN.ADS                                    --
--                                                               --
--  DATE CREATED:   02 FEB 88                                    --
--  LAST MODIFIED:  26 APR 88                                    --
--                                                               --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                     --
--                                                               --
--  DESCRIPTION:  This package defines the interface to the      --
--                token scanner module.                          --
--                                                               --
--***************************************************************--

with TEXT_IO;
package TOKEN_SCANNER is

  -- maximum number of chars per line in file being parsed
  LINESIZE : constant integer := 132;

  ENDFILE  : constant character := ASCII.sub;
  ENDLINE  : constant character := ASCII.eot;

  -- ADA token classes
  type TOKEN_CLASS is (RESERVED_WORD, IDENTIFIER, SEPARATOR, NUMERIC_LIT,
                       DELIMITER, COMMENT, CHARACTER_LIT, STRING_LIT,
                       UNDEF_CHAR, EOF);

  -- record to indicate where a token came from
  type SOURCE_RECORD is
    record
      FILE_NAME      : string(1..LINESIZE) := (others => ' ');
      FILE_NAME_SIZE : natural := 0;
      LINE_NUMBER    : natural;
    end record;

  -- record to hold the token built up by the token scanner.  the LEXEME is
  -- the actual string for that particular token and LEXEME_SIZE is the
  -- number of characters in the lexeme string.  SOURCE indicates the
  -- location in the source file where the token originated.
```

```
type TOKEN_RECORD_TYPE is
  record
    TOKEN_TYPE  : TOKEN_CLASS;
    LEXEME      : string(1..LINESIZE) := (others => ' ');
    LEXEME_SIZE : natural := 0;
    SOURCE      : SOURCE_RECORD;
  end record;

-- raising of any of the following exceptions indicates that an illegal
-- token has been scanned into the look ahead token.  In the case of an
-- exception, procedure LOOK_TOKEN is undefined, while procedure LOOK_
-- AHEAD_TOKEN can provide access to the lexeme that raised one of the
-- scanner exceptions.
ILLEGAL_IDENTIFIER  : exception;
ILLEGAL_NUMERIC_LIT : exception;
ILLEGAL_STRING_LIT  : exception;
ILLEGAL_CHARACTER   : exception;

procedure SET_UP_TOKEN_SCANNER(PARSE_FILE : in TEXT_IO.file_type);
-- pre  - must be called before any other procedure in the token
--        scanner module.  Only one file may be set up at a time.
--        PARSE_FILE must be open and rewound before token scanner
--        can be set up.

procedure RELEASE_TOKEN_SCANNER(PARSE_FILE : in out TEXT_IO.file_type);
-- pre  - TOKEN_SCANNER has been set up.
-- post - All TOKEN_SCANNER interfaces are undefined with the exception
--        of SET_UP_TOKEN_SCANNER.  The TOKEN_SCANNER must be released
--        prior to main program termination.  PARSE_FILE is closed.

procedure LOOK_TOKEN(PARSE_FILE   : in TEXT_IO.file_type;
                     TOKEN        : out TOKEN_RECORD_TYPE);
-- pre  - scanner has been set up and an exception has not occurred.
-- post - TOKEN contains the token under the read head in PARSE_FILE.
--        The scanner filters out comments and separators.

procedure LOOK_AHEAD_TOKEN(PARSE_FILE   : in TEXT_IO.file_type;
                           TOKEN        : out TOKEN_RECORD_TYPE);
-- pre  - scanner has been set up.
-- post - TOKEN contains the next token to come under the read head in
--        PARSE_FILE.  The scanner filters out comments and separators.

procedure CONSUME_TOKEN(PARSE_FILE : in TEXT_IO.file_type);
-- pre  - scanner has been set up.
-- post - the read head is advanced one token in PARSE_FILE.
--        The scanner filters out comments and separators.

function LINES_SCANNED(PARSE_FILE : in TEXT_IO.file_type) return positive;
-- pre  - scanner has been set up.
-- post - returns the number of lines in PARSE_FILE
--        that have been scanned by the token scanner.
```

```
  function COMMENTS_SCANNED(PARSE_FILE : in TEXT_IO.file_type) return natural;
  -- pre  - scanner has been set up.
  -- post - returns the number of "meaningful" comments in PARSE_FILE
  --        that have been scanned by the token scanner.  A "meaningful"
  --        comment is defined as a comment that contains at least one
  --        letter or digit.

end TOKEN_SCANNER;
```

```
--************************************************************--
--                                                            --
--  TITLE:          ADAFLOW                                   --
--                                                            --
--  MODULE NAME:    PACKAGE TOKEN_SCANNER                     --
--  FILE NAME:      TOKEN.ADB                                 --
--                                                            --
--  DATE CREATED:   02 FEB 88                                 --
--  LAST MODIFIED:  26 APR 88                                 --
--                                                            --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                  --
--                                                            --
--  DESCRIPTION:  This package contains the procedures which  --
--                implement the TOKEN_SCANNER.                --
--                                                            --
--************************************************************--


with TEXT_IO;
package body TOKEN_SCANNER is

  CURRENT_TOKEN : TOKEN_RECORD_TYPE;
  NEXT_TOKEN    : TOKEN_RECORD_TYPE;
  LINE_TOTAL    : positive := 1;
  COMMENT_TOTAL : natural  := 0;


  package BUILD_TOKEN_PIPE is

    procedure INITIALIZE_TOKEN_PIPE;

    procedure GET_TOKEN(TEXT_FILE : in TEXT_IO.file_type;
                        TOKEN     : out TOKEN_RECORD_TYPE;
                        IS_VALID  : out boolean);
  end BUILD_TOKEN_PIPE;

  package body BUILD_TOKEN_PIPE is
    subtype UPPER_CASE_LETTER is character range 'A'..'Z';
    subtype LOWER_CASE_LETTER is character range 'a'..'z';
    subtype UPPER_CASE_HEX    is character range 'A'..'F';
    subtype LOWER_CASE_HEX    is character range 'a'..'f';
    subtype DIGITS_TYPE       is character range '0'..'9';
    subtype FORMAT_EFFECTOR   is character range ASCII.HT..ASCII.CR;
    subtype CHAR_LIT_TYPE     is character range ' '..'~';
    type LOOK_UP_TABLE  is array (LOWER_CASE_LETTER) of natural;
    type STRING_MATRIX  is array (positive range 1..63) of string(1..9);

    RESERVED_WORD_MATRIX : STRING_MATRIX :=
      (("abort    "),("abs      "),("accept   "),("access   "),
       ("all      "),("and      "),("array    "),("at       "),
       ("begin    "),("body     "),("case     "),("constant "),
```

```
   ("declare  "),("delay    "),("delta    "),("digits   "),
   ("do       "),("else     "),("elsif    "),("end      "),
   ("entry    "),("exception"),("exit     "),("for      "),
   ("function "),("generic  "),("goto     "),("if       "),
   ("in       "),("is       "),("limited  "),("loop     "),
   ("mod      "),("new      "),("not      "),("null     "),
   ("of       "),("or       "),("others   "),("out      "),
   ("package  "),("pragma   "),("private  "),("procedure"),
   ("raise    "),("range    "),("record   "),("rem      "),
   ("renames  "),("return   "),("reverse  "),("select   "),
   ("separate "),("subtype  "),("task     "),("terminate"),
   ("then     "),("type     "),("use      "),("when     "),
   ("while    "),("with     "),("xor      "));

RESERVED_WORD_HASH : LOOK_UP_TABLE :=
  ((1),(9),(11),(13),(18),(24),(26),(0),(28),(0),(0),(31),(33),
  (34),(37),(41),(0),(45),(52),(55),(59),(0),(60),(63),(0),(0));
CH          : character := ' ';
CH_HOLD     : character := ' ';
INITIAL_TOKEN : boolean := TRUE;
PARTIAL_TOKEN : boolean := FALSE;
TOKEN_WAITING : boolean := FALSE;
TOKEN_HOLD : TOKEN_RECORD_TYPE;

package GET_CHAR_PIPE is
  procedure GET_CHARACTER(TEXT_FILE : in TEXT_IO.file_type;
                          CH        : out character);
end GET_CHAR_PIPE;

package body GET_CHAR_PIPE is
  procedure GET_CHARACTER(TEXT_FILE : in TEXT_IO.file_type;
                          CH        : out character) is
  begin
    if TEXT_IO.END_OF_FILE(TEXT_FILE) then
      CH := ENDFILE;
    elsif TEXT_IO.END_OF_LINE(TEXT_FILE) then
      TEXT_IO.SKIP_LINE(TEXT_FILE);
      CH := ENDLINE;
    else
      TEXT_IO.get(TEXT_FILE, CH);
    end if;
  end GET_CHARACTER;
end GET_CHAR_PIPE;

procedure INITIALIZE_TOKEN_PIPE is
begin
  CH            := ' ';
  CH_HOLD       := ' ';
  INITIAL_TOKEN := TRUE;
  PARTIAL_TOKEN := FALSE;
```

```
  TOKEN_WAITING := FALSE;
end INITIALIZE_TOKEN_PIPE;

procedure GET_TOKEN(TEXT_FILE : in TEXT_IO.file_type;
                    TOKEN     : out TOKEN_RECORD_TYPE;
                    IS_VALID  : out boolean) is
LEXEME_COUNT : positive := 1;
STATE        : positive := 1;
TEST_LEXEME  : string(1..LINESIZE);
SHARP_REPLACEMENT : boolean := FALSE;
QUOTE_REPLACEMENT : boolean := FALSE;
  function IS_RESERVED(TEST_LEXEME : in string) return boolean is
  LEXEME : string(1..9) := (others => ' ');
  IS_MATCH : boolean := FALSE;
  ROW : natural;
  INDEX_CHAR : character;
  HASH_STOP  : natural;
  begin
    if (TEST_LEXEME'LENGTH <= 9) then
      LEXEME(TEST_LEXEME'RANGE) := TEST_LEXEME;
      for I in TEST_LEXEME'RANGE loop
        if ((LEXEME(I) in DIGITS_TYPE) or else (LEXEME(I) = '_')) then
          return (FALSE);
        elsif (LEXEME(I) in UPPER_CASE_LETTER) then
          LEXEME(I) :=
            LOWER_CASE_LETTER'VAL(UPPER_CASE_LETTER'POS(LEXEME(I)) + 32);
        end if;
      end loop;
      case (LEXEME(1)) is
        when 'h'|'j'|'k'|'q'|'v'|'y'|'z' =>
          return (FALSE);
        when others =>
          ROW := RESERVED_WORD_HASH(LEXEME(1));
          if (LEXEME(1) = 'x') then
            HASH_STOP := 63;
          else
            INDEX_CHAR := character'SUCC(LEXEME(1));
            while (RESERVED_WORD_HASH(INDEX_CHAR) = 0) loop
              INDEX_CHAR := character'SUCC(INDEX_CHAR);
            end loop;
            HASH_STOP  := RESERVED_WORD_HASH(INDEX_CHAR);
          end if;
          while ((ROW <= HASH_STOP) and then (not IS_MATCH)) loop
            IS_MATCH := (LEXEME = RESERVED_WORD_MATRIX(ROW));
            ROW := ROW + 1;
          end loop;
          return (IS_MATCH);
      end case;
    else
      return (FALSE);
    end if;
```

```
  end IS_RESERVED;
begin
  TOKEN.LEXEME := (others => ' ');
  TOKEN.SOURCE.FILE_NAME := (others => ' ');
  if (INITIAL_TOKEN) then
    GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
    INITIAL_TOKEN := FALSE;
  end if;
  if ((CH /= ENDFILE) and then (not TOKEN_WAITING) and then
  (not PARTIAL_TOKEN)) then
    CH := CH_HOLD;
    GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
  elsif (PARTIAL_TOKEN) then
    PARTIAL_TOKEN := FALSE;
  end if;
  if TOKEN_WAITING then
    TOKEN := TOKEN_HOLD;
    IS_VALID := TRUE;
    TOKEN_WAITING := FALSE;
  elsif ((CH in UPPER_CASE_LETTER) or else (CH in LOWER_CASE_LETTER)) then
    TOKEN.TOKEN_TYPE := IDENTIFIER;
    TOKEN.SOURCE.LINE_NUMBER := LINE_TOTAL;
    TOKEN.SOURCE.FILE_NAME_SIZE := TEXT_IO.name(TEXT_FILE)'LENGTH;
    TOKEN.SOURCE.FILE_NAME(1..TEXT_IO.name(TEXT_FILE)'LENGTH) :=
      TEXT_IO.name(TEXT_FILE);
    TOKEN.LEXEME(LEXEME_COUNT) := CH;
    TEST_LEXEME(LEXEME_COUNT) := CH;
    loop
      case STATE is
        when 1 => if ((CH_HOLD in UPPER_CASE_LETTER) or else
                  (CH_HOLD in LOWER_CASE_LETTER) or else
                  (CH_HOLD in DIGITS_TYPE)) then
                    LEXEME_COUNT := LEXEME_COUNT + 1;
                    TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                    TEST_LEXEME(LEXEME_COUNT) := CH_HOLD;
                    GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                  elsif (CH_HOLD = '_') then
                    STATE := 2;
                    LEXEME_COUNT := LEXEME_COUNT + 1;
                    TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                    TEST_LEXEME(LEXEME_COUNT) := CH_HOLD;
                    GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                  else
                    if (IS_RESERVED(TEST_LEXEME(1..LEXEME_COUNT))) then
                      TOKEN.TOKEN_TYPE := RESERVED_WORD;
                    end if;
                    TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                    IS_VALID := TRUE;
                    exit;
                  end if;
        when 2 => if ((CH_HOLD in UPPER_CASE_LETTER) or else
```

```
                (CH_HOLD in LOWER_CASE_LETTER) or else
                (CH_HOLD in DIGITS_TYPE)) then
                  STATE := 1;
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                  TEST_LEXEME(LEXEME_COUNT) := CH_HOLD;
                  GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                else
                  IS_VALID := FALSE;
                  TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                  exit;
                end if;
      when others => null;
    end case;
  end loop;
elsif ((CH in FORMAT_EFFECTOR) or else
       (CH = ' ') or else (CH = ENDLINE)) then
  TOKEN.TOKEN_TYPE := SEPARATOR;
  TOKEN.SOURCE.LINE_NUMBER := LINE_TOTAL;
  TOKEN.SOURCE.FILE_NAME_SIZE := TEXT_IO.name(TEXT_FILE)'LENGTH;
  TOKEN.SOURCE.FILE_NAME(1..TEXT_IO.name(TEXT_FILE)'LENGTH) :=
    TEXT_IO.name(TEXT_FILE);
  TOKEN.LEXEME(LEXEME_COUNT) := CH;
  if (CH = ENDLINE) then
    LINE_TOTAL := LINE_TOTAL + 1;
  end if;
  -- go ahead and flush out the rest of the separators as they will be
  -- discarded anyway
  while ((CH_HOLD in FORMAT_EFFECTOR) or else (CH_HOLD = ' ') or else
  (CH_HOLD = ENDLINE)) loop
    LEXEME_COUNT := LEXEME_COUNT + 1;
    TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
    if (CH_HOLD = ENDLINE) then
      LINE_TOTAL := LINE_TOTAL + 1;
    end if;
    GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
  end loop;
  TOKEN.LEXEME_SIZE := LEXEME_COUNT;
  IS_VALID := TRUE;
elsif (CH in DIGITS_TYPE) then
  TOKEN.TOKEN_TYPE := NUMERIC_LIT;
  TOKEN.SOURCE.LINE_NUMBER := LINE_TOTAL;
  TOKEN.SOURCE.FILE_NAME_SIZE := TEXT_IO.name(TEXT_FILE)'LENGTH;
  TOKEN.SOURCE.FILE_NAME(1..TEXT_IO.name(TEXT_FILE)'LENGTH) :=
    TEXT_IO.name(TEXT_FILE);
  TOKEN.LEXEME(LEXEME_COUNT) := CH;
  loop
    case STATE is
      when 1 => if (CH_HOLD in DIGITS_TYPE) then
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
```

```
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        elsif (CH_HOLD = '.') then
          STATE := 2;
          LEXEME_COUNT := LEXEME_COUNT + 1;
          TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        elsif ((CH_HOLD = 'E') or else (CH_HOLD = 'e')) then
          STATE := 17;
          LEXEME_COUNT := LEXEME_COUNT + 1;
          TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        elsif (CH_HOLD = '_') then
          STATE := 9;
          LEXEME_COUNT := LEXEME_COUNT + 1;
          TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        elsif ((CH_HOLD = '#') or else (CH_HOLD = ':')) then
          SHARP_REPLACEMENT := (CH_HOLD = ':');
          STATE := 10;
          LEXEME_COUNT := LEXEME_COUNT + 1;
          TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        elsif ((CH_HOLD in UPPER_CASE_LETTER) or else (CH_HOLD in
          LOWER_CASE_LETTER)) then --must be a separator
          --between a numeric literal and an identifier.
          TOKEN.LEXEME_SIZE := LEXEME_COUNT;
          IS_VALID := FALSE;
          exit;
        else
          TOKEN.LEXEME_SIZE := LEXEME_COUNT;
          IS_VALID := TRUE;
          exit;
        end if;
when 2 => if (CH_HOLD in DIGITS_TYPE) then
          STATE := 3;
          LEXEME_COUNT := LEXEME_COUNT + 1;
          TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        elsif (CH_HOLD = '.') then  --test for range dots
          TOKEN.LEXEME(LEXEME_COUNT) := ' ';
          TOKEN.LEXEME_SIZE := LEXEME_COUNT - 1;
          IS_VALID := TRUE;
          TOKEN_HOLD.TOKEN_TYPE := DELIMITER;
          TOKEN_HOLD.LEXEME(1..2) := "..";
          TOKEN_HOLD.LEXEME_SIZE := 2;
          TOKEN_HOLD.SOURCE.LINE_NUMBER := LINE_TOTAL;
          TOKEN_HOLD.SOURCE.FILE_NAME_SIZE :=
            TEXT_IO.name(TEXT_FILE)'LENGTH;
          TOKEN_HOLD.SOURCE.FILE_NAME(1..TEXT_IO.
            name(TEXT_FILE)'LENGTH) := TEXT_IO.name(TEXT_FILE);
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
```

```
            TOKEN_WAITING := TRUE;
            exit;
          else
            TOKEN.LEXEME_SIZE := LEXEME_COUNT;
            IS_VALID := FALSE;
            exit;
          end if;
when 3 => if (CH_HOLD in DIGITS_TYPE) then
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          elsif ((CH_HOLD = 'E') or else (CH_HOLD = 'e')) then
            STATE := 4;
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          elsif (CH_HOLD = '_') then
            STATE := 5;
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          elsif ((CH_HOLD in UPPER_CASE_LETTER) or else (CH_HOLD in
            LOWER_CASE_LETTER)) then
            TOKEN.LEXEME_SIZE := LEXEME_COUNT;
            IS_VALID := FALSE;
            exit;
          else
            TOKEN.LEXEME_SIZE := LEXEME_COUNT;
            IS_VALID := TRUE;
            exit;
          end if;
when 4 => if ((CH_HOLD = '+') or else (CH_HOLD = '-')) then
            STATE := 6;
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          elsif (CH_HOLD in DIGITS_TYPE) then
            STATE := 7;
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          else
            TOKEN.LEXEME_SIZE := LEXEME_COUNT;
            IS_VALID := FALSE;
            exit;
          end if;
when 5|6|8|9 => if (CH_HOLD in DIGITS_TYPE) then
            case STATE is
              when 5   => STATE := 3;
              when 6|8 => STATE := 7;
              when 9   => STATE := 1;
```

```
            when others => null;
          end case;
          LEXEME_COUNT := LEXEME_COUNT + 1;
          TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        else
          TOKEN.LEXEME_SIZE := LEXEME_COUNT;
          IS_VALID := FALSE;
          exit;
        end if;
when 7 => if (CH_HOLD in DIGITS_TYPE) then
          LEXEME_COUNT := LEXEME_COUNT + 1;
          TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        elsif (CH_HOLD = '_') then
          STATE := 8;
          LEXEME_COUNT := LEXEME_COUNT + 1;
          TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        elsif ((CH_HOLD in UPPER_CASE_LETTER) or else (CH_HOLD in
          LOWER_CASE_LETTER)) then
          TOKEN.LEXEME_SIZE := LEXEME_COUNT;
          IS_VALID := FALSE;
          exit;
        else
          TOKEN.LEXEME_SIZE := LEXEME_COUNT;
          IS_VALID := TRUE;
          exit;
        end if;
when 10 => if ((CH_HOLD in DIGITS_TYPE) or else
        (CH_HOLD in UPPER_CASE_HEX) or else
        (CH_HOLD in LOWER_CASE_HEX)) then
          STATE := 11;
          LEXEME_COUNT := LEXEME_COUNT + 1;
          TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
        elsif ((CH_HOLD = '=') and then (SHARP_REPLACEMENT)) then
          SHARP_REPLACEMENT := FALSE;
          TOKEN.LEXEME(LEXEME_COUNT) := ' ';
          TOKEN.LEXEME_SIZE := LEXEME_COUNT - 1;
          IS_VALID := TRUE;
          TOKEN_HOLD.TOKEN_TYPE := DELIMITER;
          TOKEN_HOLD.LEXEME(1..2) := ":=";
          TOKEN_HOLD.LEXEME_SIZE := 2;
          TOKEN_HOLD.SOURCE.LINE_NUMBER := LINE_TOTAL;
          TOKEN_HOLD.SOURCE.FILE_NAME_SIZE :=
            TEXT_IO.name(TEXT_FILE)'LENGTH;
          TOKEN_HOLD.SOURCE.FILE_NAME(1..TEXT_IO.
            name(TEXT_FILE)'LENGTH) := TEXT_IO.name(TEXT_FILE);
          GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          TOKEN_WAITING := TRUE;
```

```
            exit;
          else
            TOKEN.LEXEME_SIZE := LEXEME_COUNT;
            IS_VALID := FALSE;
            exit;
          end if;
when 11 => if ((CH_HOLD in DIGITS_TYPE) or else
          (CH_HOLD in UPPER_CASE_HEX) or else
          (CH_HOLD in LOWER_CASE_HEX)) then
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          elsif (CH_HOLD = '.') then
            STATE := 14;
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          elsif (CH_HOLD = '_') then
            STATE := 12;
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          elsif (((CH_HOLD = '#') and (not SHARP_REPLACEMENT)) or
          else ((CH_HOLD = ':') and SHARP_REPLACEMENT)) then
            STATE := 13;
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          else
            TOKEN.LEXEME_SIZE := LEXEME_COUNT;
            IS_VALID := FALSE;
            exit;
          end if;
when 12|14|16 => if ((CH_HOLD in DIGITS_TYPE) or else
          (CH_HOLD in UPPER_CASE_HEX) or else
          (CH_HOLD in LOWER_CASE_HEX)) then
            case STATE is
              when 12    => STATE := 11;
              when 14|16 => STATE := 15;
              when others => null;
            end case;
            LEXEME_COUNT := LEXEME_COUNT + 1;
            TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
            GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
          else
            TOKEN.LEXEME_SIZE := LEXEME_COUNT;
            IS_VALID := FALSE;
            exit;
          end if;
when 13 => if ((CH_HOLD = 'E') or else (CH_HOLD = 'e')) then
            STATE := 17;
```

```
              LEXEME_COUNT := LEXEME_COUNT + 1;
              TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
              GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
            elsif ((CH_HOLD in UPPER_CASE_LETTER) or else (CH_HOLD in
              LOWER_CASE_LETTER)) then
              TOKEN.LEXEME_SIZE := LEXEME_COUNT;
              IS_VALID := FALSE;
              exit;
            else
              TOKEN.LEXEME_SIZE := LEXEME_COUNT;
              IS_VALID := TRUE;
              exit;
            end if;
when 15 => if ((CH_HOLD in DIGITS_TYPE) or else
            (CH_HOLD in UPPER_CASE_HEX) or else
            (CH_HOLD in LOWER_CASE_HEX)) then
              LEXEME_COUNT := LEXEME_COUNT + 1;
              TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
              GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
            elsif (CH_HOLD = '_') then
              STATE := 16;
              LEXEME_COUNT := LEXEME_COUNT + 1;
              TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
              GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
            elsif (((CH_HOLD = '#') and (not SHARP_REPLACEMENT)) or
            else ((CH_HOLD = ':') and SHARP_REPLACEMENT)) then
              STATE := 18;
              LEXEME_COUNT := LEXEME_COUNT + 1;
              TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
              GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
            else
              TOKEN.LEXEME_SIZE := LEXEME_COUNT;
              IS_VALID := FALSE;
              exit;
            end if;
when 17 => if (CH_HOLD = '+') then
              STATE := 6;
              LEXEME_COUNT := LEXEME_COUNT + 1;
              TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
              GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
            elsif (CH_HOLD in DIGITS_TYPE) then
              STATE := 7;
              LEXEME_COUNT := LEXEME_COUNT + 1;
              TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
              GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
            else
              TOKEN.LEXEME_SIZE := LEXEME_COUNT;
              IS_VALID := FALSE;
              exit;
            end if;
when 18 => if ((CH_HOLD = 'E') or else (CH_HOLD = 'e')) then
```

```
                STATE := 4;
                LEXEME_COUNT := LEXEME_COUNT + 1;
                TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
              elsif ((CH_HOLD in UPPER_CASE_LETTER) or else (CH_HOLD in
                LOWER_CASE_LETTER)) then
                TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                IS_VALID := FALSE;
                exit;
              else
                TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                IS_VALID := TRUE;
                exit;
              end if;
      when others => null;
    end case;
  end loop;
elsif (CH = ''') then
  TOKEN.SOURCE.LINE_NUMBER := LINE_TOTAL;
  TOKEN.SOURCE.FILE_NAME_SIZE := TEXT_IO.name(TEXT_FILE)'LENGTH;
  TOKEN.SOURCE.FILE_NAME(1..TEXT_IO.name(TEXT_FILE)'LENGTH) :=
    TEXT_IO.name(TEXT_FILE);
  TOKEN.LEXEME(LEXEME_COUNT) := CH;
  IS_VALID := TRUE;
  loop
    case STATE is
      when 1 => if (CH_HOLD in CHAR_LIT_TYPE) then
                STATE := 2;
                LEXEME_COUNT := LEXEME_COUNT + 1;
                TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                CH := CH_HOLD;
                GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
              else
                TOKEN.TOKEN_TYPE := DELIMITER;
                TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                exit;
              end if;
      when 2 => if (CH_HOLD = ''') then
                TOKEN.TOKEN_TYPE := CHARACTER_LIT;
                LEXEME_COUNT := LEXEME_COUNT + 1;
                TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                exit;
              else
                TOKEN.TOKEN_TYPE := DELIMITER;
                PARTIAL_TOKEN := TRUE;
                TOKEN.LEXEME(LEXEME_COUNT) := ' ';
                TOKEN.LEXEME_SIZE := LEXEME_COUNT - 1;
                exit;
              end if;
```

```
      when others => null;
    end case;
  end loop;
elsif ((CH = '&') or else (CH = '(') or else (CH = ')') or else
(CH = '*') or else (CH = '+') or else (CH = ',') or else
(CH = '-') or else (CH = '.') or else (CH = '/') or else
(CH = ':') or else (CH = ';') or else (CH = '<') or else
(CH = '=') or else (CH = '>') or else (CH = '|') or else (CH = '!')) then
  TOKEN.TOKEN_TYPE := DELIMITER;
  TOKEN.SOURCE.LINE_NUMBER := LINE_TOTAL;
  TOKEN.SOURCE.FILE_NAME_SIZE := TEXT_IO.name(TEXT_FILE)'LENGTH;
  TOKEN.SOURCE.FILE_NAME(1..TEXT_IO.name(TEXT_FILE)'LENGTH) :=
    TEXT_IO.name(TEXT_FILE);
  IS_VALID := TRUE;
  TOKEN.LEXEME(LEXEME_COUNT) := CH;
  case CH_HOLD is
    when '.' => if (CH = '.') then
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                  GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                end if;
    when '*' => if (CH = '*') then
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                  GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                end if;
    when '=' => if ((CH = ':') or else (CH = '/') or else (CH = '>') or
                else (CH = '<')) then
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                  GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                end if;
    when '>' => if ((CH = '<') or else (CH = '>') or
                else (CH = '=')) then
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                  GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                end if;
    when '<' => if (CH = '<') then
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                  GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                end if;
    when '-' => if (CH = '-') then
                  TOKEN.TOKEN_TYPE := COMMENT;
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                  GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                  while ((CH_HOLD /= ENDLINE) and
                  (CH_HOLD /= ENDFILE)) loop
                    LEXEME_COUNT := LEXEME_COUNT + 1;
```

```
                TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
              end loop;
            end if;
      when others => null;
    end case;
    TOKEN.LEXEME_SIZE := LEXEME_COUNT;
  elsif ((CH = '"') or else (CH = '%')) then
    TOKEN.TOKEN_TYPE := STRING_LIT;
    TOKEN.SOURCE.LINE_NUMBER := LINE_TOTAL;
    TOKEN.SOURCE.FILE_NAME_SIZE := TEXT_IO.name(TEXT_FILE)'LENGTH;
    TOKEN.SOURCE.FILE_NAME(1..TEXT_IO.name(TEXT_FILE)'LENGTH) :=
      TEXT_IO.name(TEXT_FILE);
    TOKEN.LEXEME(LEXEME_COUNT) := CH;
    QUOTE_REPLACEMENT := (CH = '%');
    loop
      case STATE is
        when 1 => if (((CH_HOLD = '"') and (not QUOTE_REPLACEMENT)) or else
                  ((CH_HOLD = '%') and QUOTE_REPLACEMENT)) then
                    STATE := 2;
                    LEXEME_COUNT := LEXEME_COUNT + 1;
                    TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                    GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                  elsif (CH_HOLD in CHAR_LIT_TYPE) then
                    if ((QUOTE_REPLACEMENT and (CH_HOLD /= '%')) or else
                    ((not(QUOTE_REPLACEMENT)) and (CH_HOLD /= '"'))) then
                      STATE := 4;
                      LEXEME_COUNT := LEXEME_COUNT + 1;
                      TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                      GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                    else
                      TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                      IS_VALID := FALSE;
                      exit;
                    end if;
                  else
                    TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                    IS_VALID := FALSE;
                    exit;
                  end if;
        when 2 => if (((CH_HOLD = '"') and (not QUOTE_REPLACEMENT)) or else
                  ((CH_HOLD = '%') and QUOTE_REPLACEMENT)) then
                    STATE := 3;
                    LEXEME_COUNT := LEXEME_COUNT + 1;
                    TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                    GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                  else
                    TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                    IS_VALID := TRUE;
                    exit;
                  end if;
```

```
      when 3 => if (((CH_HOLD = '"') and (not QUOTE_REPLACEMENT)) or else
                ((CH_HOLD = '%') and QUOTE_REPLACEMENT)) then
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                  TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                  GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                  exit;
                elsif (CH_HOLD in CHAR_LIT_TYPE) then
                  if ((QUOTE_REPLACEMENT and (CH_HOLD /= '%')) or else
                  ((not(QUOTE_REPLACEMENT)) and (CH_HOLD /= '"'))) then
                    STATE := 4;
                    LEXEME_COUNT := LEXEME_COUNT + 1;
                    TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                    GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                  else
                    TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                    IS_VALID := FALSE;
                    exit;
                  end if;
                else
                  TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                  IS_VALID := FALSE;
                  exit;
                end if;
      when 4 => if (((CH_HOLD = '"') and (not QUOTE_REPLACEMENT)) or else
                ((CH_HOLD = '%') and QUOTE_REPLACEMENT)) then
                  STATE := 2;
                  LEXEME_COUNT := LEXEME_COUNT + 1;
                  TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                  GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                elsif (CH_HOLD in CHAR_LIT_TYPE) then
                  if ((QUOTE_REPLACEMENT and (CH_HOLD /= '%')) or else
                  ((not(QUOTE_REPLACEMENT)) and (CH_HOLD /= '"'))) then
                    LEXEME_COUNT := LEXEME_COUNT + 1;
                    TOKEN.LEXEME(LEXEME_COUNT) := CH_HOLD;
                    GET_CHAR_PIPE.GET_CHARACTER(TEXT_FILE, CH_HOLD);
                  else
                    TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                    IS_VALID := FALSE;
                    exit;
                  end if;
                else
                  TOKEN.LEXEME_SIZE := LEXEME_COUNT;
                  IS_VALID := FALSE;
                  exit;
                end if;
      when others => null;
    end case;
  end loop;
elsif (CH = ENDFILE) then
  TOKEN.TOKEN_TYPE := EOF;
```

```
      TOKEN.SOURCE.LINE_NUMBER := LINE_TOTAL;
      TOKEN.SOURCE.FILE_NAME_SIZE := TEXT_IO.name(TEXT_FILE)'LENGTH;
      TOKEN.SOURCE.FILE_NAME(1..TEXT_IO.name(TEXT_FILE)'LENGTH) :=
        TEXT_IO.name(TEXT_FILE);
      TOKEN.LEXEME(LEXEME_COUNT) := CH;
      TOKEN.LEXEME_SIZE := LEXEME_COUNT;
      IS_VALID := TRUE;
    else  -- character is not defined in ADA
      TOKEN.TOKEN_TYPE := UNDEF_CHAR;
      TOKEN.SOURCE.LINE_NUMBER := LINE_TOTAL;
      TOKEN.SOURCE.FILE_NAME_SIZE := TEXT_IO.name(TEXT_FILE)'LENGTH;
      TOKEN.SOURCE.FILE_NAME(1..TEXT_IO.name(TEXT_FILE)'LENGTH) :=
        TEXT_IO.name(TEXT_FILE);
      TOKEN.LEXEME(LEXEME_COUNT) := CH;
      TOKEN.LEXEME_SIZE := LEXEME_COUNT;
      IS_VALID := FALSE;
    end if;
  end GET_TOKEN;
end BUILD_TOKEN_PIPE;

function VALID_COMMENT(TOKEN : in TOKEN_RECORD_TYPE) return boolean is
-- pre  - TOKEN is a comment.
-- post - if the lexeme of the comment contains at least one letter or
--        digit then VALID_COMMENT is true, else VALID_COMMENT is false.
subtype UPPER_CASE_LETTER is character range 'A'..'Z';
subtype LOWER_CASE_LETTER is character range 'a'..'z';
subtype DIGITS_TYPE       is character range '0'..'9';
IS_VALID : boolean := FALSE;
LEXEME_COUNT : positive := 3;
begin
  while ((not IS_VALID) and (LEXEME_COUNT <= TOKEN.LEXEME_SIZE)) loop
    IS_VALID := ((TOKEN.LEXEME(LEXEME_COUNT) in UPPER_CASE_LETTER) or else
                 (TOKEN.LEXEME(LEXEME_COUNT) in LOWER_CASE_LETTER) or else
                 (TOKEN.LEXEME(LEXEME_COUNT) in DIGITS_TYPE));
    LEXEME_COUNT := LEXEME_COUNT + 1;
  end loop;
  return IS_VALID;
end VALID_COMMENT;

procedure SET_UP_TOKEN_SCANNER(PARSE_FILE : in TEXT_IO.file_type) is
-- pre  - must be called before any other procedure in the TOKEN_
--        SCANNER module.  only one file may be set up at a time.
--        PARSE_FILE must be open and rewound before TOKEN_SCANNER
--        can be set up.
IS_VALID : boolean;
begin
  LINE_TOTAL := 1;
  COMMENT_TOTAL := 0;
  BUILD_TOKEN_PIPE.INITIALIZE_TOKEN_PIPE;
  BUILD_TOKEN_PIPE.GET_TOKEN(PARSE_FILE, NEXT_TOKEN, IS_VALID);
  while (IS_VALID and ((NEXT_TOKEN.TOKEN_TYPE = SEPARATOR) or else
```

```
  (NEXT_TOKEN.TOKEN_TYPE = COMMENT))) loop
    if (NEXT_TOKEN.TOKEN_TYPE = COMMENT) then
      if (VALID_COMMENT(NEXT_TOKEN)) then
        COMMENT_TOTAL := COMMENT_TOTAL + 1;
      end if;
    end if;
    BUILD_TOKEN_PIPE.GET_TOKEN(PARSE_FILE, NEXT_TOKEN, IS_VALID);
  end loop;
  if (IS_VALID) then
    CONSUME_TOKEN(PARSE_FILE);
  else
    case (NEXT_TOKEN.TOKEN_TYPE) is
      when IDENTIFIER   => raise ILLEGAL_IDENTIFIER;
      when NUMERIC_LIT  => raise ILLEGAL_NUMERIC_LIT;
      when STRING_LIT   => raise ILLEGAL_STRING_LIT;
      when UNDEF_CHAR   => raise ILLEGAL_CHARACTER;
      when others       => null;
    end case;
  end if;
end SET_UP_TOKEN_SCANNER;

procedure RELEASE_TOKEN_SCANNER(PARSE_FILE : in out TEXT_IO.file_type) is
-- pre  - TOKEN_SCANNER has been set up.
-- post - All TOKEN_SCANNER interfaces are undefined with the exception of
--        SET_UP_TOKEN_SCANNER.  The TOKEN_SCANNER must be released prior to
--        main program termination.  PARSE_FILE is closed.
begin
  TEXT_IO.close(PARSE_FILE);
end RELEASE_TOKEN_SCANNER;

procedure LOOK_TOKEN(PARSE_FILE : in TEXT_IO.file_type;
                     TOKEN      : out TOKEN_RECORD_TYPE) is
-- pre  - scanner has been set up and an exception has not occurred.
-- post - TOKEN contains the token under the read head in PARSE_FILE.
--        The scanner filters out comments and separators.
begin
        TOKEN := CURRENT_TOKEN;
end LOOK_TOKEN;

procedure LOOK_AHEAD_TOKEN(PARSE_FILE : in TEXT_IO.file_type;
                           TOKEN      : out TOKEN_RECORD_TYPE) is
-- post - TOKEN contains the next token to come under the read head in
--        PARSE_FILE.  The scanner filters out comments and separators.
begin
  TOKEN := NEXT_TOKEN;
end LOOK_AHEAD_TOKEN;

procedure CONSUME_TOKEN(PARSE_FILE : in TEXT_IO.file_type) is
-- pre  - the scanner has been set up.
-- post - the read head is advanced one token in PARSE_FILE.
--        The scanner filters out comments and separators.
```

```
  IS_VALID : boolean;
  TEMP_TOKEN : TOKEN_RECORD_TYPE;
  begin
    CURRENT_TOKEN := NEXT_TOKEN;
    if (NEXT_TOKEN.TOKEN_TYPE /= EOF) then
      BUILD_TOKEN_PIPE.GET_TOKEN(PARSE_FILE, TEMP_TOKEN, IS_VALID);
      while (IS_VALID and ((TEMP_TOKEN.TOKEN_TYPE = SEPARATOR) or else
      (TEMP_TOKEN.TOKEN_TYPE = COMMENT))) loop
        if (TEMP_TOKEN.TOKEN_TYPE = COMMENT) then
          if (VALID_COMMENT(TEMP_TOKEN)) then
            COMMENT_TOTAL := COMMENT_TOTAL + 1;
          end if;
        end if;
        BUILD_TOKEN_PIPE.GET_TOKEN(PARSE_FILE, TEMP_TOKEN, IS_VALID);
      end loop;
      if (not(IS_VALID)) then
        case (NEXT_TOKEN.TOKEN_TYPE) is
          when IDENTIFIER   => raise ILLEGAL_IDENTIFIER;
          when NUMERIC_LIT  => raise ILLEGAL_NUMERIC_LIT;
          when STRING_LIT   => raise ILLEGAL_STRING_LIT;
          when UNDEF_CHAR   => raise ILLEGAL_CHARACTER;
          when others       => null;
        end case;
      else
        NEXT_TOKEN := TEMP_TOKEN;
      end if;
    end if;
  end CONSUME_TOKEN;

  function LINES_SCANNED(PARSE_FILE : in TEXT_IO.file_type) return positive is
  -- post - returns the number of lines in PARSE_FILE
  --        that have been scanned by the token scanner.
  begin
    return CURRENT_TOKEN.SOURCE.LINE_NUMBER;
  end LINES_SCANNED;

  function COMMENTS_SCANNED(PARSE_FILE : in TEXT_IO.file_type)
  return natural is
  -- pre  - scanner has been set up.
  -- post - returns the number of "meaningful" comments in PARSE_FILE
  --        that have been scanned by the token scanner.  A "meaningful"
  --        comment is defined as a comment that contains at least one
  --        letter or digit.
  begin
    return COMMENT_TOTAL;
  end COMMENTS_SCANNED;

end TOKEN_SCANNER;
```

APPENDIX I

"ADAFLOW" PROGRAM LISTING - GENERIC PACKAGES

```
--*****************************************************************--
--                                                                 --
--  TITLE:          ADAFLOW                                        --
--                                                                 --
--  MODULE NAME:    PACKAGE GENERIC_LIST                           --
--  FILE NAME:      LIST.ADA                                       --
--                                                                 --
--  DATE CREATED:   31 MAR 88                                      --
--  LAST MODIFIED:  28 APR 88                                      --
--                                                                 --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                       --
--                                                                 --
--  DESCRIPTION:  This package defines the operations              --
--                available on the abstract data type LIST.        --
--                                                                 --
--*****************************************************************--

generic
  type ITEM_TYPE is private;
package GENERIC_LIST is

  type LIST is limited private;

  LIST_OVERFLOW  : exception;
  LIST_UNDERFLOW : exception;

-- Operations:  If the list is not empty, then one of the nodes is designated
--   as the current node.  Ocaasionally, in the postcondition, it is necessary
--   to refer to the list of the current node as they were immediately before
--   execution of the operation.  L-pre and c-pre, respectively, are employed
--   for these references.

  procedure FIND_FIRST(L : in out LIST);
  -- pre  - The list L is not empty.
  -- post - The first node is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.

  procedure FIND_NEXT(L : in out LIST);
  -- pre  - The list L is not empty and the last node is not the current node.
  -- post - c-next in L is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  --                   - LIST_OVERFLOW  if the last node is the current node.
```

```
procedure FIND_PREVIOUS(L : in out LIST);
-- pre  - The list L is not empty and the first node is not the current node.
-- post - c-prior in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty or c is the first node.

procedure FIND_LAST(L : in out LIST);
-- pre  - The list L is not empty.
-- post - The last node in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure RETRIEVE(L : in LIST; ITEM : out ITEM_TYPE);
-- pre  - The list L is not empty.
-- post - ITEM contains the value of the element in the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure UPDATE(L : in out LIST; ITEM : in ITEM_TYPE);
-- pre  - The list L is not empty.
-- post - The current node in L contains ITEM as its element.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure INSERT(L : in out LIST; ITEM : in ITEM_TYPE);
-- pre  - The number of nodes in L has not reached its bound.
-- post - A node containing ITEM is the last node in the list, and the last
--        node in L-pre, if any, is its predecessor.  The node containing
--        ITEM is the current node.
-- exceptions raised - LIST_OVERFLOW if L has reached its bound.

procedure DELETE(L : in out LIST);
-- pre  - The list L is not empty.
-- post - c-pre in not in the list L.  If c-pre was the first node,
--        then c-next, if it exists, is the successor of c-prior.  If the
--        list L is not empty, then the last node is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

function  SIZE_OF(L : in LIST) return natural;
-- post - SIZE_OF is the number of nodes in list L.

function  EMPTY(L : in LIST) return boolean;
-- post - If the list L has no nodes then EMPTY is true, else EMPTY is
--        false.

function  FULL(L : in LIST) return boolean;
-- post - If the number of nodes in the list L has reached the maximum
--        allowed, then FULL is true, else FULL is false.

function  FIRST(L : in LIST) return boolean;
-- pre  - The list L is not empty.
-- post - If the first node is the current node in L then FIRST is true, else
--        FIRST is false.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
```

```
  function  LAST(L : in LIST) return boolean;
  -- pre  - The list L is not empty.
  -- post - If the last node is the current node in L then LAST is true, else
  --        LAST is false.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.

  procedure CREATE(L : in out LIST; SUCCESS :  out boolean);
  -- post - If a list L can be created then L exists and is empty, and SUCCESS
  --        is TRUE else SUCCESS is FALSE.

  procedure DISPOSE(L : in out LIST);
  -- post - L-pre does not exist.

private

  type LIST_INSTANCE;
  type LIST is access LIST_INSTANCE;

end GENERIC_LIST;

with UNCHECKED_DEALLOCATION;
package body GENERIC_LIST is

  type NODE;
  type NODE_POINTER is access NODE;
  type NODE is
    record
      ELEMENT : ITEM_TYPE;
      NEXT    : NODE_POINTER;
    end record;
  type LIST_INSTANCE is
    record
      HEAD    : NODE_POINTER := null;
      TAIL    : NODE_POINTER := null;
      CURRENT : NODE_POINTER := null;
      SIZE    : natural := 0;
    end record;

  procedure FREE_NODE is new UNCHECKED_DEALLOCATION(NODE, NODE_POINTER);
  procedure FREE_LIST is new UNCHECKED_DEALLOCATION(LIST_INSTANCE, LIST);

  procedure FIND_FIRST(L : in out LIST) is
  -- pre  - The list L is not empty.
  -- post - The first node is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    L.CURRENT := L.HEAD;
  end FIND_FIRST;
```

```
procedure FIND_NEXT(L : in out LIST) is
-- pre  - The list L is not empty and the last node is not the current node.
-- post - c-next in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
--                   - LIST_OVERFLOW  if the last node is the current node.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  if (LAST(L)) then
    raise LIST_OVERFLOW;
  end if;
  L.CURRENT := L.CURRENT.NEXT;
end FIND_NEXT;

procedure FIND_PREVIOUS(L : in out LIST) is
-- pre  - The list L is not empty and the first node is not the current node.
-- post - c-prior in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty or c is the first node.
TEMP_POINTER : NODE_POINTER;
begin
  if (EMPTY(L) or FIRST(L)) then
    raise LIST_UNDERFLOW;
  end if;
  TEMP_POINTER := L.HEAD;
  while (TEMP_POINTER.NEXT /= L.CURRENT) loop
    TEMP_POINTER := TEMP_POINTER.NEXT;
  end loop;
  L.CURRENT := TEMP_POINTER;
end FIND_PREVIOUS;

procedure FIND_LAST(L : in out LIST) is
-- pre  - The list L is not empty.
-- post - The last node in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  while (not LAST(L)) loop
    FIND_NEXT(L);
  end loop;
end FIND_LAST;

procedure RETRIEVE(L : in LIST; ITEM : out ITEM_TYPE) is
-- pre  - The list L is not empty.
-- post - ITEM contains the value of the element in the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
```

```
  end if;
  ITEM := L.CURRENT.ELEMENT;
end RETRIEVE;

procedure UPDATE(L : in out LIST; ITEM : in ITEM_TYPE) is
-- pre  - The list L is not empty.
-- post - The current node in L contains ITEM as its element.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  L.CURRENT.ELEMENT := ITEM;
end UPDATE;

procedure INSERT(L : in out LIST; ITEM : in ITEM_TYPE) is
-- pre  - The number of nodes in L has not reached its bound.
-- post - A node containing ITEM is the last node in the list, and the last
--        node in L-pre, if any, is its predecessor.  The node containing
--        ITEM is the current node.
-- exceptions raised - LIST_OVERFLOW if L has reached its bound.
TEMP_POINTER : NODE_POINTER;
begin
  if (FULL(L)) then
    raise LIST_OVERFLOW;
  end if;
  TEMP_POINTER := new NODE'(ITEM, null);
  if (L.HEAD = null) then
    L.HEAD := TEMP_POINTER;
    L.TAIL := TEMP_POINTER;
  else
    L.TAIL.NEXT := TEMP_POINTER;
    L.TAIL      := TEMP_POINTER;
  end if;
  L.CURRENT := TEMP_POINTER;
  L.SIZE := L.SIZE + 1;
end INSERT;

procedure DELETE(L : in out LIST) is
-- pre  - The list L is not empty.
-- post - c-pre in not in the list L.  If c-pre was the first node,
--        then c-next, if it exists, is the successor of c-prior.  If the
--        list L is not empty, then the last node is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
TEMP_POINTER : NODE_POINTER;
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  if (L.CURRENT /= L.HEAD) then
    TEMP_POINTER := L.HEAD;
```

```
    while (TEMP_POINTER.NEXT /= L.CURRENT) loop
      TEMP_POINTER := TEMP_POINTER.NEXT;
    end loop;
    TEMP_POINTER.NEXT := L.CURRENT.NEXT;
    if (L.CURRENT = L.TAIL) then
      L.TAIL := TEMP_POINTER;
    end if;
  else
    if (L.HEAD = L.TAIL) then
      L.TAIL := null;
    end if;
    L.HEAD := L.HEAD.NEXT;
  end if;
  FREE_NODE(L.CURRENT);
  L.CURRENT := L.TAIL;
  L.SIZE := L.SIZE - 1;
end DELETE;

function  SIZE_OF(L : in LIST) return natural is
-- post - SIZE_OF is the number of nodes in list L.
begin
  return (L.SIZE);
end SIZE_OF;

function  EMPTY(L : in LIST) return boolean is
-- post - If the list L has no nodes then EMPTY is true, else EMPTY is
--        false.
begin
  return (L.HEAD = null);
end EMPTY;

function  FULL(L : in LIST) return boolean is
-- post - If the number of nodes in the list L has reached the maximum
--        allowed, then FULL is true, else FULL is false.
TEMP_POINTER : NODE_POINTER;
begin
  TEMP_POINTER := new NODE;
  FREE_NODE(TEMP_POINTER);
  return (FALSE);
exception
  when STORAGE_ERROR =>
    return (TRUE);
  when others =>
    raise;
end FULL;

function  FIRST(L : in LIST) return boolean is
-- pre  - The list L is not empty.
-- post - If the first node is the current node in L then FIRST is true, else
--        FIRST is false.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
```

```
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    return (L.CURRENT = L.HEAD);
  end FIRST;

  function  LAST(L : in LIST) return boolean is
  -- pre  - The list L is not empty.
  -- post - If the last node is the current node in L then LAST is true, else
  --        LAST is false.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    return (L.CURRENT = L.TAIL);
  end LAST;

  procedure CREATE(L : in out LIST; SUCCESS :  out boolean) is
  -- post - If a list L can be created then L exists and is empty, and SUCCESS
  --        is TRUE else SUCCESS is FALSE.
  begin
    L := new LIST_INSTANCE'(null, null, null, 0);
    SUCCESS := TRUE;
  exception
    when STORAGE_ERROR =>
      SUCCESS := FALSE;
    when others =>
      raise;
  end CREATE;

  procedure DISPOSE(L : in out LIST) is
  -- post - L-pre does not exist.
  begin
    if (not EMPTY(L)) then
      FIND_LAST(L);
      while (not EMPTY(L)) loop
        DELETE(L);
      end loop;
    end if;
    FREE_LIST(L);
  end DISPOSE;

end GENERIC_LIST;
```

```
--*************************************************************--
--                                                             --
--  TITLE:          ADAFLOW                                    --
--                                                             --
--  MODULE NAME:    PACKAGE ORDERED_GENERIC_LIST               --
--  FILE NAME:      ORD_LIST.ADA                               --
--                                                             --
--  DATE CREATED:   18 APR 88                                  --
--  LAST MODIFIED:  28 APR 88                                  --
--                                                             --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                   --
--                                                             --
--  DESCRIPTION:  This package defines the operations          --
--                available on the abstract data type LIST.    --
--                                                             --
--*************************************************************--

generic
  type ITEM_TYPE is private;
package ORDERED_GENERIC_LIST is

  type LIST is limited private;

  LIST_OVERFLOW  : exception;
  LIST_UNDERFLOW : exception;

-- Operations:  If the list is not empty, then one of the nodes is designated
--   as the current node.  Ocaasionally, in the postcondition, it is necessary
--   to refer to the list of the current node as they were immediately before
--   execution of the operation.  L-pre and c-pre, respectively, are employed
--   for these references.

  procedure FIND_FIRST(L : in out LIST);
  -- pre  - The list L is not empty.
  -- post - The first node is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.

  procedure FIND_NEXT(L : in out LIST);
  -- pre  - The list L is not empty and the last node is not the current node.
  -- post - c-next in L is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  --                   - LIST_OVERFLOW  if the last node is the current node.

  procedure FIND_PREVIOUS(L : in out LIST);
  -- pre  - The list L is not empty and the first node is not the current node.
  -- post - c-prior in L is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty or c is the first node.

  procedure FIND_LAST(L : in out LIST);
  -- pre  - The list L is not empty.
```

```
-- post - The last node in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure RETRIEVE(L : in LIST; ITEM : out ITEM_TYPE);
-- pre  - The list L is not empty.
-- post - ITEM contains the value of the element in the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure UPDATE(L : in out LIST; ITEM : in ITEM_TYPE);
-- pre  - The list L is not empty.
-- post - The current node in L contains ITEM as its element.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

procedure INSERT(L : in out LIST; ITEM : in ITEM_TYPE; KEY : in positive);
-- pre  - The number of nodes in L has not reached its bound.
-- post - A node containing ITEM is in the list in ascending order
--        specified by KEY.  The node containing ITEM is the current node.
-- exceptions raised - LIST_OVERFLOW if L has reached its bound.

procedure DELETE(L : in out LIST);
-- pre  - The list L is not empty.
-- post - c-pre in not in the list L.  If c-pre was the first node,
--        then c-next, if it exists, is the successor of c-prior.  If the
--        list L is not empty, then the last node is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

function  SIZE_OF(L : in LIST) return natural;
-- post - SIZE_OF is the number of nodes in list L.

function  EMPTY(L : in LIST) return boolean;
-- post - If the list L has no nodes then EMPTY is true, else EMPTY is
--        false.

function  FULL(L : in LIST) return boolean;
-- post - If the number of nodes in the list L has reached the maximum
--        allowed, then FULL is true, else FULL is false.

function  FIRST(L : in LIST) return boolean;
-- pre  - The list L is not empty.
-- post - If the first node is the current node in L then FIRST is true, else
--        FIRST is false.
-- exceptions raised - LIST_UNDERFLOW if L is empty.

function  LAST(L : in LIST) return boolean;
-- pre  - The list L is not empty.
-- post - If the last node is the current node in L then LAST is true, else
--        LAST is false.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
```

```
  procedure CREATE(L : in out LIST; SUCCESS :  out boolean);
  -- post - If a list L can be created then L exists and is empty, and SUCCESS
  --        is TRUE else SUCCESS is FALSE.

  procedure DISPOSE(L : in out LIST);
  -- post - L-pre does not exist.

private

  type LIST_INSTANCE;
  type LIST is access LIST_INSTANCE;

end ORDERED_GENERIC_LIST;

with UNCHECKED_DEALLOCATION;
package body ORDERED_GENERIC_LIST is

  type NODE;
  type NODE_POINTER is access NODE;
  type NODE is
    record
      KEY     : positive;
      ELEMENT : ITEM_TYPE;
      NEXT    : NODE_POINTER;
    end record;
  type LIST_INSTANCE is
    record
      HEAD    : NODE_POINTER := null;
      TAIL    : NODE_POINTER := null;
      CURRENT : NODE_POINTER := null;
      SIZE    : natural := 0;
    end record;

  procedure FREE_NODE is new UNCHECKED_DEALLOCATION(NODE, NODE_POINTER);
  procedure FREE_LIST is new UNCHECKED_DEALLOCATION(LIST_INSTANCE, LIST);

  procedure FIND_FIRST(L : in out LIST) is
  -- pre  - The list L is not empty.
  -- post - The first node is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
  begin
    if (EMPTY(L)) then
      raise LIST_UNDERFLOW;
    end if;
    L.CURRENT := L.HEAD;
  end FIND_FIRST;

  procedure FIND_NEXT(L : in out LIST) is
  -- pre  - The list L is not empty and the last node is not the current node.
  -- post - c-next in L is the current node.
  -- exceptions raised - LIST_UNDERFLOW if L is empty.
```

```
--                    - LIST_OVERFLOW  if the last node is the current node.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  if (LAST(L)) then
    raise LIST_OVERFLOW;
  end if;
  L.CURRENT := L.CURRENT.NEXT;
end FIND_NEXT;

procedure FIND_PREVIOUS(L : in out LIST) is
-- pre  - The list L is not empty and the first node is not the current node.
-- post - c-prior in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty or c is the first node.
TEMP_POINTER : NODE_POINTER;
begin
  if (EMPTY(L) or FIRST(L)) then
    raise LIST_UNDERFLOW;
  end if;
  TEMP_POINTER := L.HEAD;
  while (TEMP_POINTER.NEXT /= L.CURRENT) loop
    TEMP_POINTER := TEMP_POINTER.NEXT;
  end loop;
  L.CURRENT := TEMP_POINTER;
end FIND_PREVIOUS;

procedure FIND_LAST(L : in out LIST) is
-- pre  - The list L is not empty.
-- post - The last node in L is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  while (not LAST(L)) loop
    FIND_NEXT(L);
  end loop;
end FIND_LAST;

procedure RETRIEVE(L : in LIST; ITEM : out ITEM_TYPE) is
-- pre  - The list L is not empty.
-- post - ITEM contains the value of the element in the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  ITEM := L.CURRENT.ELEMENT;
end RETRIEVE;
```

```
procedure UPDATE(L : in out LIST; ITEM : in ITEM_TYPE) is
-- pre  - The list L is not empty.
-- post - The current node in L contains ITEM as its element.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  L.CURRENT.ELEMENT := ITEM;
end UPDATE;

procedure INSERT(L : in out LIST; ITEM : in ITEM_TYPE; KEY : in positive) is
-- pre  - The number of nodes in L has not reached its bound.
-- post - A node containing ITEM is in the list in ascending order
--        specified by KEY.  The node containing ITEM is the current node.
-- exceptions raised - LIST_OVERFLOW if L has reached its bound.
TEMP_POINTER   : NODE_POINTER;
SEARCH_POINTER : NODE_POINTER;
begin
  if (FULL(L)) then
    raise LIST_OVERFLOW;
  end if;
  TEMP_POINTER := new NODE'(KEY, ITEM, null);
  if (L.HEAD = null) then
    L.HEAD := TEMP_POINTER;
    L.TAIL := TEMP_POINTER;
  else
    if (L.HEAD.KEY > KEY) then
      TEMP_POINTER.NEXT := L.HEAD;
      L.HEAD := TEMP_POINTER;
    else
      SEARCH_POINTER := L.HEAD.NEXT;
      if (SEARCH_POINTER /= null) then
        if (SEARCH_POINTER.KEY > KEY) then
          TEMP_POINTER.NEXT := SEARCH_POINTER;
          L.HEAD.NEXT := TEMP_POINTER;
        else
          while ((SEARCH_POINTER.NEXT /= null) and then
                 (SEARCH_POINTER.NEXT.KEY < KEY)) loop
            SEARCH_POINTER := SEARCH_POINTER.NEXT;
          end loop;
          TEMP_POINTER.NEXT := SEARCH_POINTER.NEXT;
          SEARCH_POINTER.NEXT := TEMP_POINTER;
          if (SEARCH_POINTER = L.TAIL) then
            L.TAIL := TEMP_POINTER;
          end if;
        end if;
      else
        L.HEAD.NEXT := TEMP_POINTER;
        L.TAIL := TEMP_POINTER;
      end if;
```

```
    end if;
  end if;
  L.CURRENT := TEMP_POINTER;
  L.SIZE := L.SIZE + 1;
end INSERT;

procedure DELETE(L : in out LIST) is
-- pre  - The list L is not empty.
-- post - c-pre in not in the list L.  If c-pre was the first node,
--        then c-next, if it exists, is the successor of c-prior.  If the
--        list L is not empty, then the last node is the current node.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
TEMP_POINTER : NODE_POINTER;
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  if (L.CURRENT /= L.HEAD) then
    TEMP_POINTER := L.HEAD;
    while (TEMP_POINTER.NEXT /= L.CURRENT) loop
      TEMP_POINTER := TEMP_POINTER.NEXT;
    end loop;
    TEMP_POINTER.NEXT := L.CURRENT.NEXT;
    if (L.CURRENT = L.TAIL) then
      L.TAIL := TEMP_POINTER;
    end if;
  else
    if (L.HEAD = L.TAIL) then
      L.TAIL := null;
    end if;
    L.HEAD := L.HEAD.NEXT;
  end if;
  FREE_NODE(L.CURRENT);
  L.CURRENT := L.TAIL;
  L.SIZE := L.SIZE - 1;
end DELETE;

function  SIZE_OF(L : in LIST) return natural is
-- post - SIZE_OF is the number of nodes in list L.
begin
  return (L.SIZE);
end SIZE_OF;

function  EMPTY(L : in LIST) return boolean is
-- post - If the list L has no nodes then EMPTY is true, else EMPTY is
--        false.
begin
  return (L.HEAD = null);
end EMPTY;
```

```
function  FULL(L : in LIST) return boolean is
-- post - If the number of nodes in the list L has reached the maximum
--        allowed, then FULL is true, else FULL is false.
TEMP_POINTER : NODE_POINTER;
begin
  TEMP_POINTER := new NODE;
  FREE_NODE(TEMP_POINTER);
  return (FALSE);
exception
  when STORAGE_ERROR =>
    return (TRUE);
  when others =>
    raise;
end FULL;

function  FIRST(L : in LIST) return boolean is
-- pre  - The list L is not empty.
-- post - If the first node is the current node in L then FIRST is true, else
--        FIRST is false.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  return (L.CURRENT = L.HEAD);
end FIRST;

function  LAST(L : in LIST) return boolean is
-- pre  - The list L is not empty.
-- post - If the last node is the current node in L then LAST is true, else
--        LAST is false.
-- exceptions raised - LIST_UNDERFLOW if L is empty.
begin
  if (EMPTY(L)) then
    raise LIST_UNDERFLOW;
  end if;
  return (L.CURRENT = L.TAIL);
end LAST;

procedure CREATE(L : in out LIST; SUCCESS :  out boolean) is
-- post - If a list L can be created then L exists and is empty, and SUCCESS
--        is TRUE else SUCCESS is FALSE.
begin
  L := new LIST_INSTANCE'(null, null, null, 0);
  SUCCESS := TRUE;
exception
  when STORAGE_ERROR =>
    SUCCESS := FALSE;
  when others =>
    raise;
end CREATE;
```

```
  procedure DISPOSE(L : in out LIST) is
  -- post - L-pre does not exist.
  begin
    if (not EMPTY(L)) then
      FIND_LAST(L);
      while (not EMPTY(L)) loop
        DELETE(L);
      end loop;
    end if;
    FREE_LIST(L);
  end DISPOSE;

end ORDERED_GENERIC_LIST;
```

```
--*************************************************************--
--                                                             --
--  TITLE:          ADAFLOW                                    --
--                                                             --
--  MODULE NAME:    PACKAGE GENERIC_STACK                      --
--  FILE NAME:      STACK.ADA                                  --
--                                                             --
--  DATE CREATED:   31 MAR 88                                  --
--  LAST MODIFIED:  28 APR 88                                  --
--                                                             --
--  AUTHOR(S):      LT ALBERT J. GRECCO, USN                   --
--                                                             --
--  DESCRIPTION:  This package defines the operations          --
--                available on the abstract data type STACK.   --
--                                                             --
--*************************************************************--

generic
  type ITEM_TYPE is private;
package GENERIC_STACK is

  type STACK is limited private;

  STACK_OVERFLOW  : exception;
  STACK_UNDERFLOW : exception;

  procedure POP(S : in out STACK; ITEM : out ITEM_TYPE);
  -- pre  - The stack S is not empty.
  -- post - ITEM contains the most recently arrived element of S-pre.
  --        S no longer contains ITEM.
  -- exceptions raised - STACK_UNDERFLOW if S is empty.

  procedure TOP(S : in STACK; ITEM : out ITEM_TYPE);
  -- pre  - The stack S is not empty.
  -- post - ITEM contains the most recently arrived element of S-pre.
  -- exceptions raised - STACK_UNDERFLOW if S is empty.

  procedure PUSH(S : in out STACK; ITEM : in ITEM_TYPE);
  -- pre  - The size of S has not reached its bound.
  -- post - S includes ITEM as its most recently arrived element.
  -- exceptions raised - STACK_OVERFLOW if S has reached its bound.

  function  EMPTY(S : in STACK) return boolean;
  -- post - If the stack S has no ITEMS then EMPTY is true, else EMPTY is
  --        false.

  function  FULL(S : in STACK) return boolean;
  -- post - If the number of ITEMS in the stack S has reached the maximum
  --        allowed, then FULL is true, else FULL is false.
```

```
  procedure CREATE(S : in out STACK; SUCCESS :  out boolean);
  -- post - If a stack S can be created then S exists and is empty, and SUCCESS
  --        is TRUE else SUCCESS is FALSE.

  procedure DISPOSE(S : in out STACK);
  -- post - S-pre does not exist.

private

  type NODE;
  type STACK is access NODE;

end GENERIC_STACK;

with UNCHECKED_DEALLOCATION;
package body GENERIC_STACK is

  type NODE is
    record
      ELEMENT : ITEM_TYPE;
      NEXT    : STACK;
    end record;

  procedure FREE_NODE is new UNCHECKED_DEALLOCATION(NODE, STACK);

  procedure POP(S : in out STACK; ITEM : out ITEM_TYPE) is
  -- pre  - The stack S is not empty.
  -- post - ITEM contains the most recently arrived element of S-pre.
  --        S no longer contains ITEM.
  -- exceptions raised - STACK_UNDERFLOW if S is empty.
  TEMP_POINTER : STACK;
  begin
    if (EMPTY(S)) then
      raise STACK_UNDERFLOW;
    end if;
    ITEM := S.ELEMENT;
    TEMP_POINTER := S;
    S := S.NEXT;
    FREE_NODE(TEMP_POINTER);
  end POP;

  procedure TOP(S : in STACK; ITEM : out ITEM_TYPE) is
  -- pre  - The stack S is not empty.
  -- post - ITEM contains the most recently arrived element of S-pre.
  -- exceptions raised - STACK_UNDERFLOW if S is empty.
  begin
    if (EMPTY(S)) then
      raise STACK_UNDERFLOW;
    end if;
    ITEM := S.ELEMENT;
  end TOP;
```

```
procedure PUSH(S : in out STACK; ITEM : in ITEM_TYPE) is
-- pre  - The size of S has not reached its bound.
-- post - S includes ITEM as its most recently arrived element.
-- exceptions raised - STACK_OVERFLOW if S has reached its bound.
TEMP_POINTER : STACK;
begin
  if (FULL(S)) then
    raise STACK_OVERFLOW;
  end if;
  TEMP_POINTER := new NODE'(ITEM, S);
  S := TEMP_POINTER;
end PUSH;

function  EMPTY(S : in STACK) return boolean is
-- post - If the stack S has no ITEMS then EMPTY is true, else EMPTY is
--        false.
begin
  return (S = null);
end EMPTY;

function  FULL(S : in STACK) return boolean is
-- post - If the number of ITEMS in the stack S has reached the maximum
--        allowed, then FULL is true, else FULL is false.
TEMP_POINTER : STACK;
begin
  TEMP_POINTER := new NODE;
  FREE_NODE(TEMP_POINTER);
  return (FALSE);
exception
  when STORAGE_ERROR =>
    return (TRUE);
  when others =>
    raise;
end FULL;

procedure CREATE(S : in out STACK; SUCCESS :  out boolean) is
-- post - If a stack S can be created then S exists and is empty, and SUCCESS
--        is TRUE else SUCCESS is FALSE.
begin
  S := null;
  SUCCESS := TRUE;
end CREATE;

procedure DISPOSE(S : in out STACK) is
-- post - S-pre does not exist.
TEMP_POINTER : STACK;
begin
  while (S /= null) loop
    TEMP_POINTER := S;
    S := S.NEXT;
    FREE_NODE(TEMP_POINTER);
```

```
    end loop;
  end DISPOSE;

end GENERIC_STACK;
```

LIST OF REFERENCES

1. Nieder, J. L., and Fairbanks, K. S., AdaMeasure: An Ada® Software Metric, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1987.

2. Herzig, P. M., AdaMeasure: An Implementation of the Halstead and Henry Metrics, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1987.

3. Leveson, N. G., and Stolzy, J. L., "Safety Analysis Using Petri Nets", IEEE Transactions on Software Engineering, v. SE-13, No. 3, pp.386-397, March 1987.

4. Shatz, S. M., and Cheng, W. K., "An Approach to Automated Static Analysis of Distributed Software", Proceedings of the First International Conference on Supercomputing Systems, St. Petersburg, Florida, pp. 377-385, December, 1985.

5. Shatz, S. M., "On Complexity Metrics Oriented for Distributed Programs Using Ada® Tasking", Proceedings of COMPSAC-86, Chicago, Illinois, pp. 247-253, October, 1986.

6. Petri, C. A., Kommunikation mit Automaten, Ph. D. Dissertation, University of Bonn, Bonn, West Germany, 1962.

7. Peterson, J. L., Petri Net Theory and the Modeling of Systems, Englewood Cliffs, N. J., Prentice-Hall, 1981.

8. Dennis, J. and Van Horn, E., "Programming Semantics for Multiprogrammed Computations", Communications of the ACM, v. 9, No. 3, pp. 143-155, March, 1966.

9. Dijkstra, E., "Cooperating Sequential Processes", in F. Genuys(Editor), Programming Languages, New York: Academic Press, pp. 43-112, 1968.

10. Dijkstra, E., "Solution of a Problem in Concurrent Program Control", Communications of the ACM, v. 8, No. 9, p. 569, September, 1965.

11. Courtois, P., Heymans, F., and Parnas, D.,"Concurrent Control with 'Readers' and 'Writers'", Communications of the ACM, v. 14, No. 10, pp. 667-668, October, 1971.

12. Department of Defense Military Standard ANSI/MIL-STD-1815A, Ada® Programming Language, 22 January 1983.

Ada is a registered trademark of the U. S. Government--Ada Joint Program Office

13. Barrett, W. A., and others, Compiler Construction: Theory and Practice, 2d ed., Science Research Associates, Inc., 1986.

14. Department of Information and Computer Science, University of California, Irvine Technical Report #86-25, A Guided Tour of P-NUT (Petri Net UTilities Release 2.2), R. R. Razouk, January 1987.

15. Department of Information and Computer Science, University of California, Irvine Technical Report #87-04, RGA Users Manual (Version 2.3), E. Timothy Morgan, January 1987.

16. Lewis, A., Petri Net Modeling and Automated Software Safety Analysis: Methodology for an Embedded Military System, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1988.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, California 93943	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	Prof. Daniel L. Davis MBARI 160 Central Avenue Pacific Grove, California 93950	2
6.	Prof. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
7.	LCDR John M. Yurchak, USN, Code 52Yu Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
8.	Center of Naval Analysis 2000 N. Beauregard Street Alexandria, Virginia 22311	1
9.	Dr. Ralph Wachter Office of Naval Research Arlington, Virginia 22217-5000	1

10. Mr. Robert Westbrook — 1
CMDR, Code 33181
Naval Weapons Center
China Lake, California 93555

11. Mr. Carl Hall — 1
Software Missile Branch, Code 3922
Naval Weapons Center
China Lake, California 93555

12. LT Karl S. Fairbanks, Jr., USN — 1
Software Missile Branch, Code 3922
Naval Weapons Center
China Lake, California 93555

13. LT Albert J. Grecco, USN — 2
Naval Surface Weapon Systems Engineering Station
Port Hueneme, California 93043

14. Mr. Joel Trimble — 1
STARS Program Office
OUSDR&E
1211 South Fern Street
Arlington, Virginia 22202

15. Prof. Nancy Leveson — 1
Department of Information and Computer Science
University of California
Irvine, California 92717

Thesis
G7247 Grecco
c.1 AdaFlow: the automation
of software analysis
using Petri nets.